VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Nástroj pro verifikaci Workflow sítě
Software for Workflow Verification

2009 Lukáš Potenský

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7.5.2009 ……

Abstrakt

O pojmu workflow se poprvé hovoří v roce 1980. Od té doby se z něj stává velmi často skloňovaný termín užívaný při každodenní práci ve větších společnostech. Tato práce se zabývá jednak možnostmi specifikace workflow pomocí vybrané části modelovacího jazyka UML 2.0 a to konkrétněji pomocí podmnožiny prvků diagramu aktivit jazyka UML 2.0. Dále také pojednává o možnostech verifikace takto specifikovaného workflow, a to nejprve o semiformální verifikaci pomocí definovaných pravidel pro diagram aktivit UML 2.0, a následně také o možnosti formální verifikace specifikovaného workflow užitím převodu workflow na Workflow síť, která je rozšířením Petriho sítí, a její následnou verifikací pomocí formálních metod popsaných v teorii Petriho sítí. Ve druhé části se práce zabývá implementací nástroje pro editaci workflow pomocí vybrané části modelovacího jazyka UML 2.0 a její verifikaci pomocí semiformálních a formálních metod. Detailně rozebírá možnost použití Eclipse GEF pro vytváření popsaného grafického nástroje a také použití Eclipse RCP jako platformy pro samostatnou aplikaci.

Klíčová slova

workflow, Workflow síť, Petriho síť, formální verifikace, graf dosažitelnosti, spolehlivost, UML, diagram aktivit, Eclipse platform, Eclipse GEF, Eclipse RCP

Abstract

Conception of workflow was found in 1980. Since that time it has became very frequently used term in everyday work in bigger companies. This thesis deals with possibilities of workflow specification using chosen subsection of UML 2.0 modeling language, more precisely it deals with modeling of workflow using subset of UML 2.0 activity diagram elements. It also describes possibilities of verification of such defined workflows, firstly it is mentioned possibility of semiformal verification using defined UML 2.0 activity diagram rules and secondly it is mentioned how to possibly do formal verification of specified workflow using workflow transformation into Workflow net, an extension of Petri net, and subsequent verification of such Workflow net using formal methods described in Petri net theory. Implementation of workflow tool for editing workflow based on chosen subset of UML 2.0 modeling language and for its semiformal and formal verification is described in the second part. It describes in detail possibility to use Eclipse GEF as implementation framework for such graphical editor tool and also describes how to use Eclipse RCP platform for standalone application.

Keywords

workflow, Workflow net, Petri net, formal verification, reachability graph, soundness, UML, activity diagram, Eclipse platform, Eclipse GEF, Eclipse RCP

Seznam použitých symbolů a zkratek

Eclipse GEF	- Eclipse Graphical Editing Framework
Eclipse RCP	- Eclipse Rich Client Platform
JAXB	- Java Architecture for XML Binding
UML	- Unified Modeling Language
WF síť	- Workflow síť

Obsah

Seznam obrázků

Seznam příloh

1. Úvod

V této diplomové práci se zaměřuji na možnosti specifikace byznys procesů, potažmo workflow, z hlediska aktivit, událostí a podmínek, za kterých mohou být jednotlivé aktivity prováděny, a to pomocí vybrané podmnožiny jazyka UML 2.0. Konkrétněji se tedy budu zabývat specifikací workflow pomocí přístupu specifikace chování využitím modelu tvořeného striktně specifikovanou podmnožinou prvků diagramu aktivit jazyka UML 2.0. Objasním navržená syntaktická a sémantická pravidla semiformální verifikace tohoto modelu, objasním navržený postup převodu tohoto modelu na formální model postavený na Workflow sítích a možnosti jeho následné formální verifikace.

1.1. Struktura práce

V druhé kapitole seznámím čtenáře obecně s problematikou byznys modelování, metod při něm využívaných, jejich výhod a nedostatků, jejich rozdělení a vyjasním, kterou metodu jsem v této práci zvolil a proč. Následně v kapitole třetí teoreticky rozvedu podstatu diagramu aktivit jazyka UML 2.0 a možnosti jak pomocí podmnožiny jeho prvků specifikovat workflow. Uvedu zde také možnosti semiformální verifikace takto specifikovaného workflow a konkrétní postup implementace obecného balíčku tříd reprezentující tento model. Ve čtvrté kapitole nastíním část teorie Petriho sítí a Workflow sítí a jejich vlastností. Rozeberu možnosti formální specifikace workflow pomocí Workflow sítí, analýzu jejich vlastností a způsoby formální verifikace. Popíši algoritmy použité při této analýze a formální verifikaci a jejich implementaci. Zmíním se také o postupu implementace obecného balíčku tříd reprezentující tento formální model. Následně v práci představím myšlenku a postup převodu semiformálního modelu workflow specifikovaného pomocí podmnožiny prvků diagramu aktivit jazyka UML 2.0 na formální model Workflow sítě, implementaci tohoto převodu a možnosti zpětné vazby, to vše v kapitole páté.

Kapitola šestá již pojednává o vizi a problematice návrhu a implementace samotného nástroje k editaci a verifikaci workflow využívajícího dříve implementovaných vytvořených balíčků obou modelů a implementace jejich převodu. Popisuje také použité technologie, frameworky a platformy. Podrobnější pohled na samotný editor využívající platformy Eclipse GEF, jeho části a jeho implementaci nabízí kapitola sedmá. Předposlední, osmá, kapitola pak pojednává o využití platformy Eclipse RCP k vytvoření samostatné aplikace tohoto nástroje nezávislé na platformě.

2. Byznys modelování

Byznys modelování je v dnešní době stále ještě na počátku svého vývoje a jeho masivní nasazení do praxe je otázkou blízké budoucnosti. Již dnes je však o různé metody byznys modelování velký zájem. Ten vychází jednak z potřeb softwarových inženýrů, kteří jej cíleně využívají v počáteční fázi softwarového procesu tvorby nového informačního systému či jakéhokoliv jiného softwarového díla. Softwaroví inženýři využívají byznys modelování za účelem správného pochopení doménové oblasti, pro kterou je softwarové dílo vytvářeno, vzájemného porozumění a usnadnění komunikace mezi nimi a zadavateli projektu v otázkách doménové oblasti, ale také jako osvědčený prostředek k nalezení odpovědi na otázku, které specifikované doménové procesy lze automatizovat implementací funkcí nově vznikajícího softwarového díla, a tak identifikovat případně rozšířit přidanou hodnotu vnikajícího díla pro zadavatele. Z druhé strany je o různé metody byznys modelování velký zájem v oblasti podnikového managementu, kde se tyto metody využívají k popisu, tedy specifikaci byznys procesů uvnitř podniku, jejich následnou analýzu, případnou optimalizaci a především pak k řízení a automatizaci těchto byznys procesů.

2.1. Nástroje

Tímto narážíme na celou řadu nástrojů pro zpracování byznys procesů, které se dají rozdělit na 3 hlavní skupiny. BPR (Business Process Re-engineering) jsou nástroje určené k modelování a analýze byznys procesů, jejich cílem je umožnit radikálně, nebo postupně procesy vylepšovat a umožnit podle nich vlastní řízení organizace či podniku. ERP (Enterprise Resource Planning) systémy jako SAP, BAAN, Oracle apod. umožňují automatizovat výrobní procesy, finanční toky a řídit lidské zdroje, právě na základě explicitně popsaných procesů. WFM (Workflow Management) systémy reprezentující generické softwarové nástroje pro definici, správu, realizaci a vlastní řízení podnikových procesů. [1]

2.2. Byznys proces versus workflow

Neustále zde používám pojmu byznys proces, a proto by bylo vhodné jej korektně definovat. Dovolím si využít již známé a zavedené definice. *Byznys proces je po částech uspořádaná množina procedur a aktivit, které společně realizují podnikatelský nebo strategický cíl, obvykle v kontextu organizační struktury definující funkce rolí a jejich vztahy. Model byznys procesu se pak dá chápat jako abstrakce konkrétního byznys procesu použitelná pro další automatizované zpracování.[1]* Dále v textu bych rád využíval dvou termínů, a to workflow a workflow model. Workflow a byznys proces jsou zaměnitelné pojmy, jsou v podstatě totožné, jediným rozdílem

mezi nimi je fakt, že workflow je byznys proces, který je spravován či řízen ERP či WFM nástrojem. Jelikož se dále v textu budu zabývat byznys procesy z hlediska zpracování softwarovými nástroji, přejdu do této roviny plynule z obecné roviny a využiji terminologie workflow. Pojmem workflow model pak budu poukazovat na abstraktní model konkrétního workflow.

2.3. Metody specifikace byznys procesů

Metod jak specifikovat a modelovat workflow je celá řada. Co však mají všechny tyto metody společné, je určitý abstraktní rámec, který vychází z postupu návrhu byznys procesu. Nejprve je nutné identifikovat, jaké funkce daná organizace či podnik má plnit a za jakým účelem. Hledáme co je vstupem a výstupem těchto funkcí a jak jsou tyto funkce strukturovány. Následuje další krok popisující jak tyto funkce budou zajišťovat transformaci vstupů na výstupy pomocí k tomu určených aktivit a procesů. Nakonec je nutné definovat, čím konkrétně jsou tyto toky, definované v předchozích krocích, dané, a kdo a co bude realizovat specifikované aktivity. Z těchto tří uvedených základních pohledů na modelovaný byznys proces vyplývá, že existují tři základní přístupy k modelování byznys procesů, které dohromady tvoří zmiňovaný abstraktní rámec. *[1]* Jsou to tyto:

1. Funkční přístup zaměřený především na funkce, jejich strukturování, vstupy a výstupy.
2. Přístup specifikací chování je zaměřen na řídící aspekt vykonávání procesu cestou stanovení událostí a podmínek, za kterých mohou být jednotlivé aktivity prováděny.
3. Strukturální přístup je zaměřen na statický aspekt procesu. Cílem je postihnout entity a zdroje vystupující v procesu včetně jejich atributů, činností (služeb) a vzájemných vazeb.

2.4. Formální versus neformální metody

Všechny moderní metody modelování byznys procesů se dále dají rozdělit na dvě hlavní skupiny, a to na metody formální a neformální. Neformální metody specifikace workflow nám umožňují modelovaný workflow popsat často snadnými, strukturovanými, dokonce i standardizovanými nástroji, avšak postrádají přesný popis syntaxe a sémantiky. Naopak formální metody specifikace workflow mají syntaxi a sémantiku precizně popsánu a umožní nám tak modelovaný workflow popsat naprosto jednoznačně. Nespornou výhodou formálních metod specifikace workflow je také fakt, že formální metody nejsou primárně určeny pouze k jednoznačné specifikaci workflow, ale také k jejich následné analýze a verifikaci. Díky formálním metodám tak získáváme možnost ověřit si vlastnosti specifikovaného workflow. Problémem formálních metod specifikace workflow je náročnost, úsilí a čas potřebný k vytvoření modelu. Samozřejmě existují systémy, u kterých je nekompromisně vyžadována bezchybná

funkcionalita za všech podmínek. A právě pro specifikaci těchto systémů jsou formální metody využívány. U běžných systémů, u kterých se časová investice na celkovou formální specifikaci nevyplatí, lze pomocí formálních metod analyzovat a verifikovat alespoň jejich nejdůležitější části.

2.5. Semiformální metody

Mezistupněm mezi formálními a neformálními metodami specifikace workflow jsou semiformální metody. Výhodou těchto metod je, že mají přesně definovanou syntaxi. Avšak nám neumožní popsat modelovaný systém zcela jednoznačně, neboť nemají komplexně a precizně popsánu sémantiku. Z hlediska použití jsou však tyto metody velmi zajímavé. Nabízejí totiž výhody neformálních metod, tedy jednoduchost a časovou nenáročnost, a také částečně výhodu formálních metod v určité míře verifikovat specifikovaný workflow model. *Mezi semiformální metody řadíme například metody IDEF (Integration Definition), EPC (Event-driven Process Chain) i metodu specifikace workflow pomocí UML (Unified Modeling Language). [1]* Každá z nich má určité výhody a nevýhody a tím se také liší využití těchto metod v různých nástrojích.

2.5.1. Metoda IDEF

Tato metoda je dobře formalizována, má jednoznačně danou syntaxi a i sémantika je dobře specifikována, přesto lze u sémantiky nalézt některé neurčitosti v interpretaci, co je skutečně vstupem, mechanismem a řízením. Funkční analýza obsahuje všechny důležité aspekty procesu a díky rozpracovanému způsobu strukturování a přijatým konvencím lze pomocí IDEF0 popisovat i velmi složité (komplexní) procesy. Metoda je standardizována na úrovni National Institute of Standards and Technology. IDEF0 popisuje pouze funkce a zanedbává skutečný průběh procesu, tedy jak jsou jednotlivé aktivity za sebou řazeny z hlediska času. I když je možné tuto posloupnost z diagramů IDEF0 částečně odvodit, není tento náhled jejich prioritou. Komplexní popis procesu lze realizovat s využitím dalších technik (např. již zmíněný IDEF1, IDEF2, IDEF3 atd.), pak se však celá metoda stává náročnou z hlediska jejího zvládnutí a ztrácí se názornost, která je primárním požadavkem modelování obecně. [1]

2.5.2. Metoda EPC

EPC Tato metoda poskytuje jednoduchý princip spojení událostí a aktivit usnadňující vytváření i velmi složitých procesů. EPC diagramy jsou základním nástrojem popisu procesů u celé řady komerčně úspěšných a v masovém měřítku nasazovaných softwarových systémů jako jsou SAP R/3, ARIS, LiveModel/Analyst a Microsoft Visio. Jazyk, který je v EPC používán není formálně definovaný, syntaxe ani sémantika není důsledně dána, což může vést k nejednoznačnosti ve specifikaci procesů. Situace může být o to horší, že se předpokládá

softwarová implementace těchto procesů v podobě workflow v rámci ERP a WFM systémů. Nejednoznačnosti či chybné specifikace mohou pak vést k problémům při jejich vykonávání. Nemusí být zaručeno dosažení požadovaného koncového stavu procesu z důvodu jeho uvíznutí čekáním na nesplnitelnou podmínku nebo nekonečném opakování nějakého z cyklů obsaženém v procesu. Chybějící formální specifikace omezuje možnosti použití procesů specifikovaných v EPC v jiných produktech, než výše uvedených. Je tak omezena přenositelnost mezi jednotlivými produkty. [1]

2.5.3. Metoda specifikace byznys procesů pomocí UML

Jazyk UML poskytuje širokou škálu diagramů umožňujících, díky různým typům použitých abstrakcí, kompletní popis i velmi složitých byznys procesů. Notace jazyka UML je standardizována a je součástí velké řady softwarových produktů určených k modelování systémů, navíc je nespornou výhodou, že některé z těchto produktů jsou volně k dispozici. Přijetí jazyka UML za de facto standard při vývoji softwarových systémů zajišťuje návaznost vytvořených byznys modelů na specifikaci požadavků a vlastní návrh implementace informačních systémů. Komunity byznys orientovaných odborníků a softwarových inženýrů tak mohou bez zásadních problémů komunikovat pomocí společného, a především všem srozumitelného, jazyka. Bohužel má jazyk UML i své nevýhody, které jsou velmi podobné těm, které jsou vlastní i metodě EPC. I když se autoři jazyka a celá řada organizací pracujících na jeho standardizaci snaží o formalizaci UML, zatím nelze tento jazyk považovat za formálně definovaný a mohou tak nastat problémy s odhalováním chyb ve specifikovaných procesech. [1]

Jazyk UML se dá tedy považovat za všeobecně uznávaný standard, má obrovské expresivní schopnosti, přitom je relativně jednoduchý a především je používán širokým spektrem odborníků z oboru informatiky. V této diplomové práci se tedy konkrétně zaměřím na metodu specifikace byznys procesů pomocí jazyka UML 2.0. Abych byl přesnější, zaměřím se na specifikaci byznys procesů z pohledu jejich aktivit, událostí a podmínek, za kterých mohou být jednotlivé aktivity prováděny, tedy pomocí přístupu specifikace chování pomocí jazyka UML 2.0. Takto úzce vymezený pohled je řešen ze strany UML 2.0 pomocí diagramů aktivit.

3. Diagram aktivit UML 2.0

Diagram aktivit jazyka UML 2.0 je volně definovaný typ vývojového diagramu popisujícího chování, jehož účelem je specifikovat pořadí toku činností jednotlivých kroků. Zachycuje dynamiku modelu pomocí vykonávaných aktivit reprezentujících jeho vnitřní stavy a pomocí řídících toků reprezentujících přechody mezi těmito stavy způsobené ukončením těchto aktivit. Obsahuje prostředky pro modelování výběru, opakování a paralelismu. Obecně se diagram aktivit jazyka UML 2.0 používá nejen k modelování průběhu jednotlivých Use Case v softwarovém procesu, ale také v byznys modelování ke specifikaci byznys procesů a workflow. *[3,4]*

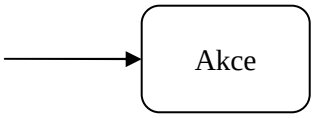
3.1. Modelování workflow pomocí diagramu aktivit UML 2.0

Výhody specifikace a modelování byznys procesů, potažmo workflow pomocí jazyka UML jsem již shrnul v kapitole 2.5.3. V této diplomové práci se zaměřím na specifikaci a modelování workflow z pohledu jejich aktivit, událostí a podmínek, za kterých mohou být jednotlivé aktivity prováděny, tedy pomocí přístupu specifikace chování pomocí diagramů aktivit jazyka UML 2.0. Prvky modelu tohoto diagramu však nabízí širší modelovací schopnosti. Diagram aktivit UML 2.0 dokáže mimo jednotlivých aktivit, událostí, podmínek a řídících toků zachytit také objekty, které jsou aktivitami produkovány, nebo jsou jejími vstupy, dále také datové toky, kterými jsou objekty transportovány mezi aktivitami, a v neposlední řadě role, které jsou za vykonávání aktivit zodpovědné. V této práci však této modelovací dimenze diagramů aktivit UML 2.0 nevyužiji a zaměřím se pouze na řídící rovinu vykonávání aktivit workflow.

3.2. Prvky diagramu aktivit UML 2.0 pro modelování workflow

Ke specifikaci a konstrukci modelu diagramu aktivit definuje UML 2.0 sadu základních prvků a jejich symbolů. Pro modelování workflow jsem však zvolil určitou podmnožinu těchto prvků, především těch které souvisí s dynamickým aspektem modelovaného workflow. V následujícím textu se pokusím popsat syntaxi a sémantiku jednotlivých prvků.

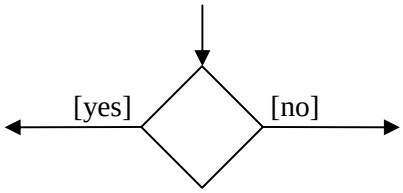
3.2.1. Akce (Action node)



Obrázek 1 – akce s příchozím řídícím tokem dle UML 2.0

Akce je pojmenovaná, nejprimitivnější a dále nedekomponovatelná aktivita workflow. Do akce může vstupovat libovolná množina příchozích řídících toků a z akce může vystupovat nejvýše jeden odchozí řídící tok. Akce je spuštěna okamžitě, když je aktivován libovolný řídící tok z množiny jejich příchozích řídících toků. Po vykonání akce je aktivován její odchozí řídící tok, pokud takový existuje.

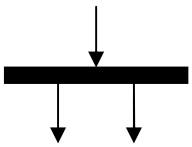
3.2.2. Rozhodovací blok (Decision node)



Obrázek 2 – rozhodovací blok s řídícími toky dle UML 2.0

Rozhodovací blok patří do skupiny řídících prvků diagramu aktivit UML 2.0. Řídící prvky jsou zodpovědné za koordinaci řídících toků mezi dalšími prvky diagramu aktivit UML 2.0. Do rozhodovacího bloku může vstupovat libovolná množina příchozích řídících toků a z rozhodovacího bloku může vystupovat libovolná množina odchozích řídících toků s různými rozhodovacími podmínkami. Rozhodovací blok je proveden okamžitě, když je aktivován libovolný řídící tok z množiny jeho příchozích řídících toků, přičemž na základě definovaných rozhodovacích podmínek řídících toků z množiny jeho odchozích řídících toků je vybrán jeden konkrétní odchozí řídící tok, který je aktivován, pokud takový existuje.

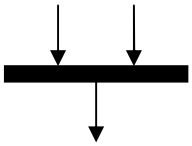
3.2.3. Rozdělovací synchronizační blok (Fork node)



Obrázek 3 – rozdělovací synchronizační blok s řídícími toky dle UML 2.0

Rozdělovací synchronizační blok také patří do skupiny řídících prvků diagramu aktivit UML 2.0. Do rozdělovacího synchronizačního bloku může vstupovat libovolná množina příchozích řídících toků a z rozdělovacího synchronizačního bloku může vystupovat libovolná množina odchozích řídících toků. Rozdělovací synchronizační blok je proveden okamžitě, když je aktivován libovolný řídící tok z množiny jeho příchozích řídících toků, přičemž jsou aktivovány všechny řídící toky z množiny jeho odchozích řídících toků, které jsou poté zpracovávány paralelně.

3.2.4. Slučovací synchronizační blok (Join node)



Obrázek 4 – spojovací synchronizační blok s řídícími toky dle UML 2.0

Slučovací synchronizační blok také patří do skupiny řídících prvků diagramu aktivit UML 2.0. Do slučovacího synchronizačního bloku může vstupovat libovolná množina příchozích řídících toků a ze slučovacího synchronizačního bloku může vystupovat nejvýše jeden odchozí řídící tok. Slučovací synchronizační blok je proveden okamžitě, když jsou aktivovány všechny řídící toky z množiny jeho příchozích řídících toků, přičemž je aktivován jeho odchozí řídící tok, pokud takový existuje.

3.2.5. Počáteční uzel (Activity initial node)

Obrázek 5 – počáteční uzel s odchozím řídícím tokem dle UML 2.0

Počáteční uzel také patří do skupiny řídících prvků diagramu aktivit UML 2.0. Do počátečního uzlu nemůže vstupovat žádný příchozí řídící tok a z počátečního uzlu může vystupovat nejvýše jeden odchozí řídící tok. Počáteční uzel je proveden okamžitě po spuštění celé aktivity popisované modelem, přičemž je aktivován jeho odchozí řídící tok, pokud takový existuje.

3.2.6. Koncový uzel (Activity final node)

Obrázek 6 – koncový uzel s příchozím řídícím tokem dle UML 2.0

Koncový uzel také patří do skupiny řídících prvků diagramu aktivit UML 2.0. Do koncového uzlu může vstupovat libovolná množina příchozích řídících toků a z koncového uzlu nemůže vystupovat žádný odchozí řídící tok. Koncový uzel je proveden okamžitě, když je aktivován libovolný řídící tok z množiny jeho příchozích řídících toků, přičemž je okamžitě ukončena celá aktivita popisovaná modelem.

3.2.7. Řídicí tok (Control flow)

Obrázek 7 – řídící tok a řídící tok s podmínkou dle UML 2.0

Řídící tok je abstraktem hrany mezi prvky diagramu aktivit UML 2.0. Řídící tok má právě jeden zdroj a právě jeden cíl. Zdrojem i cílem mohou být pouze akce nebo řídící prvky diagramu aktivit UML 2.0. Řídící tok může mít dodatečně specifikovanou podmínku, za které může být aktivován.

3.3. Verifikace diagramů aktivit UML 2.0

Nyní máme popsány všechny základní stavební prvky a jejich symboly pro modelování workflow a můžeme se zaměřit na otázku jak dále výsledný model těmito prvky tvořený verifikovat. *Bohužel je nutné zdůraznit fakt, že model diagramu aktivit UML 2.0 není plně formalizovaný z pohledu sémantiky a proto nelze provést formální verifikaci takovéhoto modelu.[2]* Přesto však tato metoda specifikace workflow patří mezi semiformální metody a tudíž lze určité vlastnosti modelu analyzovat a také je verifikovat. Naprosto triviálním řešením verifikace by mohla být důsledná vizuální kontrola vytvořeného diagramu aktivit. Toto řešení má nesporně široké spektrum nevýhod. I při vizuální kontrole nemusíme odhalit všechny možné chyby modelu a navíc je téměř nemožné vizuálně verifikovat komplexnější workflow s přítomností mnoha paralelismů, rozhodovacích bloků či cyklů. Díky dříve specifikované jednoznačné sémantice jednotlivých základních prvků modelu je však tento proces možné zautomatizovat a některé vlastnosti analyzovat programaticky.

3.4. Semiformální verifikace diagramu aktivit UML 2.0

V této podkapitole popíši vybrané a definované vlastnosti a pravidla, které by měl mít model diagramu aktivit UML 2.0, aby jej bylo možno považovat za semiformálně korektní. Je to v podstatě množina pravidel známých ze specifikace diagramu aktivit UML 2.0 rozšířená o pravidla, která jsou přenesena z požadavků na vlastnosti workflow.

3.4.1. Přítomnost právě jednoho počátečního uzlu

Aby byl model workflow specifikovaný pomocí diagramů aktivit semiformálně korektní musí obsahovat právě jeden počáteční uzel. Toto pravidlo není dle mého názoru dále popisovat. Je zřejmé, že každý workflow má právě jeden jasně specifikovaný počátek. Pokud by workflow neměl specifikovaný svůj počátek, nebylo by jej možné nikdy zahájit.

3.4.2. Přítomnost alespoň jednoho koncového uzlu

Aby byl model workflow specifikovaný pomocí diagramů aktivit semiformálně korektní musí obsahovat alespoň jeden koncový uzel. Toto pravidlo také vychází z praktické představy o vlastnostech workflow. Každý workflow by měl konkrétně definovaný svůj konec, ve kterém je ukončen. Pokud by workflow neměl specifikovaný žádný koncový uzel, nebylo by jej možné nikdy korektně ukončit.

3.4.3. Neexistence nedosažitelných uzlů

Semiformálně korektní model workflow nesmí obsahovat žádný nedosažitelný uzel (prvek). Pokud tedy budeme považovat uzly modelu za vrcholy grafu, řídící toky modelu za orientované hrany grafu a počáteční uzel modelu za výchozí vrchol grafu, pak musí být splněna podmínka, že každý vrchol takto vytvořeného grafu musí být z výchozího vrcholu dosažitelný. Tedy všechny vrcholy grafu tvoří jednu souvislou komponentu.

3.4.4. Parita synchronizačních bloků

Celkem velkým a komplexním problémem semiformálně korektního modelu workflow je pak parita rozdělovacích a spojovacích synchronizačních bloků, pomocí kterých je možné modelovat paralelně prováděné akce. Abychom při modelování workflow dosáhli semiformálně korektního modelu musíme, zjednodušeně řečeno, zajistit, aby všechny větve řídících toků vycházejících z rozdělovacího synchronizačního uzlu byly na sobě nezávislé a všechny opět vcházely řídícím tokem do spojovacího synchronizačního uzlu spárovaného s tímto rozdělovacím synchronizačním uzlem. Tímto je zajištěno hned několik podstatných faktů. Především je ihned zřejmé, že počet řídících toků vycházejících z rozdělovacího synchronizačního uzlu je roven počtu řídících toků vcházejících do spojovacího synchronizačního uzlu spárovaného s tímto rozdělovacím synchronizačním uzlem, což je v souladu se sémantikou diagramu aktivit UML 2.0. Nezávislost jednotlivých větví tvořených řídících toků vycházejících z rozdělovacího synchronizačního uzlu pak zajišťuje, že akce probíhající v jedné takovéto větvi nemůže prolnout do jiné větve a nemůže tak dojít k deadlocku na slučovacím synchronizačnímu uzlu.

Pokud celou tuto problematiku převedu do teorie grafu, snadno a jednoznačně popíši jednotlivá pravidla jak vyplývají z výše definovaných požadavků na semiformální korektnost modelovaného workflow. Nechť tedy každému rozdělovacímu synchronizačnímu uzlu v modelu přísluší právě jeden slučovací synchronizační uzel a naopak. Mějme graf, jehož vrcholy jsou tvořeny základní prvky diagramu aktivit mimo řídících toků, ty tvoří orientované hrany tohoto grafu. Pak pro všechny vrcholy synchronizačních bloků tohoto grafu můžeme definovat následující pravidla. Každá cesta vycházející z vrcholu rozdělovacího synchronizačního uzlu

musí vést do vrcholu jemu příslušného slučovacího synchronizačního uzlu. Tím je zajištěno, že z rozdělovacího synchronizačního bloku nemůžeme dojít do koncového uzlu, tedy workflow nemůže být ukončen mezi synchronizačními bloky, kde jsou paralelně zpracovávány různé akce. Pokud existuje cesta vycházející z vrcholu rozdělovacího synchronizačního uzlu a končící opět v tomto vrcholu, pak tato musí obsahovat vrchol slučovacího synchronizačního uzlu spárovaného s tímto rozdělovacím synchronizačním uzlem. Tímto je zajištěno, že neexistuje cyklus, pomocí něhož bychom mohli dojít z rozdělovacího synchronizačního bloku opět do stejného rozdělovacího synchronizačního bloku, tedy v modelu neexistuje takovýto cyklus produkující lifelock. A nakonec uvažujme pouze podgraf tohoto grafu tvořený všemi vrcholy, kterými prochází libovolná cesta vycházející z vrcholu rozdělovacího synchronizačního uzlu a končící ve vrcholu jemu příslušného slučovacího synchronizačního uzlu. Tento podgraf je pak nutně tvořen souvislými komponentami, jejichž počet musí být roven počtu řídících toků vycházejících z rozdělovacího synchronizačního uzlu. Tímto je zajištěna nezávislost jednotlivých větví tvořených řídících toků vycházejících z rozdělovacího synchronizačního uzlu.

3.5. Model diagramu aktivit UML 2.0

Výše popsánu množinu použitých prvků diagramu aktivit pro specifikaci workflow, jejich sémantiku a následně algoritmus pro semiformalní verifikaci takového modelu podle navržených pravidel jsem implementoval co nejobecněji tak, aby byl výsledný balíček použitelný i mimo tuto diplomovou práci.

3.5.1. Návrh a implementace

Nejdříve jsem tedy navrhnul množinu dobře definovaných rozhraní, které dohromady odrážejí všechny vlastnosti množiny zvolených prvků diagramu aktivit. Základním rozhraním pro všechny prvky je **IActivityDiagramElement** obsahující **getId()** a **setId()** metody pro práci s ID prvku. Z tohoto rozhraní dědí další rozhraní **IActivityDiagramNode**, které je předkem rozhraní všech uzlů diagramu aktivit, tedy **IActionNode**, **IDecisionNode**, **IForkNode**, **IJoinNode**, **IActivityInitialNode** a **IActivityFinalNode**. Z rozhraní **IActivityDiagramElement** dědí také rozhraní **IControlFlow** a **IConditionalControlFlow** reprezentující řídící datový tok a řídící datový tok s podmínkou. Dále jsem definoval dvě rozhraní popisující schopnosti uzlů diagramu aktivit být zdrojem jednoho či více odchozích řídících datových toků a také dvě rozhraní popisující schopnosti uzlů diagramu aktivit být cílem pro jeden či více příchozích řídících datových toků. Tato rozhraní jsou: **ISingleControlFlowSource**, **IMultipleControlFlowSource**, **ISingleControlFlowTarget** a **IMultipleControlFlowTarget**. Takže například rozhraní **IActionNode** reprezentující akci diagramu aktivit dědí z rozhraní **IActivityDiagramNode**, **IMultipleControlFlowTarget** a **ISingleControlFlowSource**, neboť se jedná o uzel, který do kterého

může vstupovat libovolná množina řídících toků a ze kterého může vystupovat právě jen řídící tok. Následně jsem definoval rozhraní diagramu aktivit **IActivityDiagram**, které definuje všechny veřejně dostupné metody pro práci s diagramem aktivit. Tedy metody pro přidávání, odebírání a získávání prvků modelu a metodu pro zpracování semiformální verifikace modelu.

Po dokončení návrhu tohoto modelu jsem výše popsána rozhraní doplnil implementačními třídami, které tato rozhraní implementují. Implementace je obecná a nezávislá na dalších částech této práce. Za zmínku stojí snad algoritmus přidělování unikátních ID jednotlivým prvkům, který využívá aktuálních časových razítek, v čase vytvoření instance objektu a následnému ověření zda takovéto ID se již nenachází v množině existujících vydaných unikátních ID, která je implementována jako statická proměnná třídy **ActivityDiagramElement**, která je supertřídou všech tříd reprezentující prvky diagramu aktivit. Největší implementační třídou se pochopitelně stala samotná třída reprezentující model diagramu aktivit **ActivityDiagram**. a to především metody sloužící k semiformální verifikaci tohoto modelu.

3.5.2. Algoritmus semiformální verifikace

Implementaci tohoto algoritmu jsem rozložil do několika logických částí - kroků. Některé z těchto částí využívají jednoduché či složitější známé algoritmy z teorie grafu. V prvním kroku algoritmus testuje přítomnost počátečního uzlu v modelu. Ve druhém kroku je otestována přítomnost alespoň jednoho koncového uzlu v modelu. Následuje algoritmus založený na procházení grafu do šířky, který z počátečního uzlu projde všechny uzly grafu dosažitelné řídícími toky. Tyto projité uzly označí a porovná je s množinou všech uzlů grafu a rozhodne, zda je množina nedosažitelných uzlů prázdná. Pokud množina nedosažitelných uzlů prázdná není, je vyhozena výjimka a tato množina je předána jako reference pro pozdější případné zpracování, které se například může hodit při vizualizaci těchto nedosažitelných uzlů.

Posledním krokem je test parity synchronizačních uzlů modelu, který se dá také rozložit do jednotlivých kroků. Nejprve algoritmus projde množinu všech rozdělovacích a spojovacích synchronizačních uzlů modelu a testuje, zda každý rozdělovací synchronizační blok má přiřazen spojovací synchronizační blok, který zpětně odkazuje opět na tento rozdělovací synchronizační blok a naopak. Pokud je nalezena chyba algoritmus končí, vyhodí výjimku s referencí na chybový synchronizační blok. Pokud jsou všechny synchronizační bloky v pořádku, následuje další krok, který verifikuje jednotlivé větve řídících toků vycházejících z rozdělovacího synchronizačního bloku. Tato část algoritmu je založena na procházení grafu do hloubky. Pro každý zpracovávaný rozdělovací synchronizační blok je vytvořena prázdná množina, do které se budou ukládat již projité uzly. Pro každý odchozí řídící tok zpracovávaného rozdělovacího synchronizačního bloku je pak vytvořena další prázdná množina, do které se budou ukládat již projité uzly větve tohoto řídícího toku. Od uzlu, do kterého pak tento řídící tok vchází je pak

prováděno procházení grafu do hloubky. Každý nově navštívený uzel je otestován, zda není koncovým uzlem nebo zda se nejedná o samotný rozdělovací synchronizační blok, ze kterého jsme vyšli, pak algoritmus končí vyhozením výjimky. Pokud je tento nově navštívený uzel slučovacím synchronizačním blokem, který je spárován s rozdělovacím synchronizačním blokem, ze kterého jsme vyšli, pak je zpracovávaná větev v pořádku a můžeme testovat další. Pokud se nejedná o žádný takový uzel, je takový uzel otestován, zda se již nenachází v množině již projitých uzlů zpracovávané větve. Pokud již takový uzel v této množině je, pak je tato cesta v pořádku a můžeme pokračovat v prohledávání do hloubky. Pokud takový uzel v množině není, pak je nutné otestovat, zda je tento uzel v množině všech projitých uzlů. Nachází-li se tam, pak je tento uzel společný dvěma různým větvím, algoritmus končí a je vyhozena výjimka s referencí na tento uzel. Nenachází-li se tento uzel v množině všech projitých uzlů, pak se vezmou všechny jeho odchozí řídící toky, jejichž cílové uzly jsou vloženy do zásobníku pro další zpracování. Pokud však žádný takový uzel neexistuje, pak je porušen sled akcí mezi synchronizačními bloky a celý algoritmus končí vyhozením výjimky s referencí na aktuální uzel, ze kterého již nelze pokračovat dále.

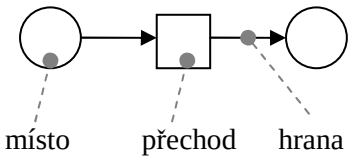
4. Petriho a Workflow sítě

V předchozí kapitole jsem navrhnul způsob jak model workflow specifikovaný pomocí diagramu aktivit UML 2.0 verifikovat semiformálně a popsal verifikační algoritmus. S touto ne zcela formální metodou verifikace se však nemůžeme spokojit a tak se v této kapitole zaměřím na formální metodu verifikace workflow pomocí sítí Workflow postavených na teorii Petriho sítí.

4.1. Popis a teorie Petriho sítí

Autorem teorie Petriho sítí je Carl Adam Petri, který ji publikoval poprvé v roce 1962 ve své disertační práci. Tato teorie je v podstatě rozšířením teorie konečných automatů. Velmi zjednodušeně vyjádřeno Petriho sítě řeší velmi elegantně problematiku parciálních stavů konečných automatů, tedy stavů, ve kterých je nedeterministický konečný automat zároveň v několika různých stavech, a to pomocí převedení modelovaní přechodů nedeterministického konečného automatu na přechody Petriho sítě. Z jednoho přechodu Petriho sítě je pak možné přejít do libovolné množiny míst, vyjadřující stavy nedeterministického konečného automatu, zároveň. *[5,6]*

Jak už jsem naznačil, narozdíl od konečných automatů, mají Petriho sítě tři základní stavební prvky. Jsou to místa (places), přechody (transitions) a hrany (arcs) Petriho sítě, kde hrany mohou vést pouze z místa do přechodu a z přechodu do místa. Dvě místa nebo dva přechody nemohou být hranou propojeny. Místo, do kterého vede hrana z nějakého přechodu se nazývá výstupní místo tohoto přechodu. Místo, ze kterého vede hrana do nějakého přechodu se nazývá vstupní místo tohoto přechodu. *[5,6]*



Obrázek 8 – základní prvky Petriho sítě

Dalším důležitým pojmy v teorii Petriho sítí jsou značka (token) a značení sítě (marking). Značka je úzce spjata s místem Petriho sítě. Místo může obsahovat libovolný nezáporný počet značek. Významem této značky na místě Petriho sítě je v podstatě splnění podmínky, kterou toto místo vyjadřuje. Značení sítě je pak možné chápat jako konkrétní rozložení značek v místech dané Petriho sítě. Konkrétně teoreticky vyjádřeno – značení sítě je funkce, která každému místu dané Petriho sítě přiřadí nezáporné číslo rovnající se počtu značek na tomto místě. S těmito pojmy pak také souvisí celá dynamika Petriho sítí. Přechody Petriho sítě mohou přemísťovat

značky z jejich vstupních míst do jejich výstupních míst, tzv. provedení přechodu (transition firing). Přechod je proveditelný pouze za předpokladu, že se na všech jeho vstupních místech nachází alespoň jedna značka. Při provedení přechodu je z každého vstupního místa tohoto přechodu značka odebrána a následně je na všechna výstupní místa tohoto přechodu přidána právě jedna značka. *[5,6]*

4.2. Graf dosažitelnosti (reachability graph)

Při provedení proveditelného přechodu Petriho sítě přecházíme z jednoho znační $\mathbf{M_1}$ na druhé značení $\mathbf{M_2}$, o kterém lze říci, že je dosažitelné ze značení $\mathbf{M_1}$. Tato dosažitelnost je však definována induktivně, tedy značení $\mathbf{M_1}$ je dosažitelné ze značení $\mathbf{M_2}$ tehdy, existuje je-li posloupnost proveditelných přechodů Petriho sítě při značení $\mathbf{M_2}$ takových, že po jejich provádění postupně získáváme další značení a v posledním kroku získáme značení $\mathbf{M_1}$. Při modelování workflow nás právě zajímá, zda je např. koncový stav dosažitelný z počátečního stavu, tedy převedeno do terminologie Petriho sítě, zda je značení reprezentující konečný stav workflow v jeho modelu dosažitelné ze značení reprezentujícího počáteční stav modelovaného workflow. Na tuto a podobné otázky dosažitelnosti snadno nalezneme odpovědi v grafu dosažitelnosti (reachability graph) modelovaného workflow. Jednotlivé uzly tohoto grafu jsou tvořeny značeními, které jsou dosažitelné z počátečního značení a orientované hrany mezi dvěma značeními reprezentují přechody, jejichž provedením bylo dosaženo značení do kterého tato hrana vede. *[5]*

Algoritmus konstrukce grafu dosažitelnosti zcela vychází z jeho definice. Jedná se v podstatě o simulaci modelované Petriho sítě. Vychází se z počátečního značení Petriho sítě. Při konstrukci jsou použity dvě množiny, první množina **P** je množinou značení určených ke zpracování, druhá množina **R** je množina již dosažených značení. V prvním kroku je do množiny **P** značení určených ke zpracování přidáno výchozí značení. V druhém kroku je z množiny **P** značení určených ke zpracování vyjmuto libovolné značení $\mathbf{M_0}$ a je přidáno do množiny **R** již dosažených značení a jsou určeny všechny přímo dosažitelná značení $\mathbf{M_X}$ provedením všech proveditelných přechodů při značení $\mathbf{M_0}$. Každé takto přímo dosažitelné značení $\mathbf{M_X}$, které ještě není v množině dosažených značení **R**, je přidáno do množiny **P** značení určených ke zpracování. Druhý krok se opakuje dokud není množina **P** značení určených ke zpracování prázdná.

Algoritmus konstrukce grafu dosažitelnosti je v celku jednoduchý, avšak skrývá jeden potenciální problém. Tento problém vychází z teoreticky nekonečné množiny dosažitelných značení dané Petriho sítě. Je velice snadné demonstrovat Petriho síť, která bude mít nekonečně mnoho dosažitelných značení a tedy nekonečný graf dosažitelnosti. Důvodem existence této spočetné nekonečné množiny dosažitelných značení určité Petriho sítě je existence určitého

místa, které v cyklu produkuje značky z tohoto cyklu odcházející. Existují samozřejmě algoritmy, které dokáží takovýto cyklus snadno odhalit na základě pokrývání značení v tomto cyklu a vydat odpověď, že graf dosažitelnosti je nekonečný. V tomto okamžiku ale také dochází k tomu, že díky nedeterministickému provádění přechodů v Petriho síti nelze dále tento i částečně sestrojený graf dosažitelnosti využít k analýze vlastností této Petriho sítě. Existuje však ještě jedno řešení, které nám umožní výpočet potenciálně nekonečného grafu dosažitelnosti omezit na základě vlastnosti, která se nazývá omezenost Petriho sítě. Tento algoritmus je založen na původním algoritmu pro konstrukci grafu dosažitelnosti s tím, že k dalšímu zpracování řadí pouze ta značení, jejichž místa mají počet značek menší nebo roven zvolené konstantě **n**. Pokud algoritmus narazí při zpracovávání na značení, jehož některé místo má více značek než zvolená konstanta **n**, je toto místo označeno za neomezené a dále již není algoritmem zpracováváno. Tento algoritmus nám po určitém množství iterací, dle zvolené konstanty **n**, dává alespoň částečný náhled na vlastnosti Petriho sítě, prostřednictvím jejího zkonstruovaného grafu dosažitelnosti.

4.3. Workflow sítě

Workflow síť je v podstatě rozšířením teorie Petriho sítí o jistá fakta vyplývající z požadavků kladených na workflow samotné, za účelem modelování workflow pomocí Petriho sítí. Co je tedy těmito fakty, která nejsou zcela zachycena klasickou Petriho sítí? Základním triviálním příkladem je začátek a konec workflow. Každý správně definovaný workflow musí mít specifikovaný začátek a konec a proto při jeho specifikaci pomocí Petriho sítí musíme být schopni toto zachytit. K tomuto účelu rozšíříme Petriho síť o definici počátečního a koncového místa Workflow sítě. Počáteční místo Workflow sítě, je místo, ve kterém celý workflow začíná. Koncové místo Workflow sítě je analogicky místo, kde celý workflow končí. Každá správně specifikovaná Workflow síť bude obsahovat právě jedno počáteční a právě jedno koncové místo. Na začátku zpracování workflow se předpokládá, že na jeho počátečním místě Workflow sítě modelující tento workflow bude umístěna značka, tzv. počáteční značení (initial marking). Během vykonávání workflow se provádějí jednotlivé přechody Workflow sítě modelující tento workflow a značky jsou tak umísťovány na různá místa této sítě. V okamžiku ukončení workflow je značka umístěna na koncovém místě Workflow sítě modelující tento workflow a dále by již neměla být na žádném místě značka příslušející právě dokončeného workflow, to by znamenalo, že je workflow špatně navržen. *[7]*

4.4. Vlastnosti Petriho a Workflow sítí

Při procesu formální verifikace workflow pomocí Workflow sítí využijeme vlastností Petriho sítí, konkrétně omezenosti a živosti Petriho sítě, a vlastnosti Workflow sítí, konkrétně spolehlivosti Workflow sítě.

4.4.1. Omezenost (boundeness)

Omezenost je obecně vlastnost Petriho sítí. Omezenost Petriho sítě určuje nezáporné přirozené číslo **n** takové, že pro všechna značení dosažitelná z počátečního značení platí, že na každém místě tohoto značení je počet značek menší nebo roven tomuto číslu **n**. Pokud pro danou Petriho síť takové přirozené číslo existuje, pak mluvíme o omezené Petriho síti. Přesněji lze hovořit o **n**-omezené Petriho síti. Speciálním případem omezené Petriho sítě je pak bezpečná Petriho síť, pro kterou platí, že toto číslo **n** je menší nebo rovno 1.

Je vhodné poznamenat, že pokud je Petriho síť omezená, pak má také konečný graf dosažitelnosti, neboť existuje pouze konečná množina permutací počtu značek na jednotlivých místech v sítí a tedy i konečná množina značení.

Nejsnadnější algoritmus rozhodující zda je Petriho síť **n**-omezená je rozšířením algoritmu pro konstrukci grafu dosažitelnosti s omezující konstantou **n**. Tento algoritmus je schopen určit, zda je Petriho síť **n**-omezená. Pokud je při konstrukci grafu dosažitelnosti překročena zvolena konstanta **n**, pak je zřejmé, že taková Petriho síť není **n**-omezená, avšak pak již není tento algoritmus schopen rozhodnout, zda je tato Petriho síť omezená, což je základní nevýhodou tohoto algoritmu.

4.4.2. Živost (liveness)

Živost je také vlastností Petriho sítí. Velmi zjednodušeně živost Petriho sítě ukazuje, zda je možné provádět všechny přechody v konkrétní Petriho síti opakovaně libovolný mnohokrát. Živost je tedy vlastnost Petriho sítě poukazující na znovu-opakovatelnost kterékoliv její části. Pokud přeneseme tuto vlastnost do roviny modelovaného workflow, pak tato vlastnost zaručuje, že modelovaný workflow je znovu-opakovatelný. Tedy pokud je Petriho síť živá, pak neobsahuje žádné deadlocky ani lifelocky. Pokud si pak představíme graf dosažitelnosti živé Petriho sítě, pak se jedná o orientovaný graf bez vrcholů, ze kterých nevede žádná hrana. V tomto grafu jsou tedy pouze vrcholy, jejichž výstupní stupeň je větší nebo roven 1. Pozor, zde musím upozornit, že toto tvrzení je ve tvaru implikace. Tedy pokud je Petriho síť živá, obsahuje její graf dosažitelnosti pouze vrcholy, jejichž výstupní stupeň je větší nebo roven 1. A tedy se této vlastnosti grafu dosažitelnosti Petriho sítě nedá využít v algoritmu rozhodujícím, zda je daná Petriho síť živá.

Definujme tedy formálně živost Petriho sítě. Značení **M** Petriho sítě je živé, jestliže je dosažitelné z počátečního značení $\mathbf{M_0}$ a zároveň pro všechny přechody **t** této petriho sítě existuje značení $\mathbf{M_X}$ dosažitelné ze značení **M** takové, že přechod **t** je v tomto značení $\mathbf{M_X}$ proveditelný. Jsou-li všechna značení **M** Petriho sítě dosažitelná z počátečního značení $\mathbf{M_0}$ živá, pak je celá tato Petriho síť živá.

Algoritmů, rozhodujících zda je daná Petriho síť živá, existuje celá řada. Já zde uvedu algoritmus založený na backtracingu. Nevýhodou backtrackingových algoritmů je v obecnosti jejich exponenciální časová složitost. Tento prezentovaný algoritmus je však založen na backtrackingovém procházení grafu dosažitelnosti do hloubky tak, že každý uzel je projit pouze jedenkrát a tedy jeho časová složitost není zdaleka exponenciální a pro naši aplikaci zcela vyhovuje. Na začátku algoritmu je zvoleno jedno libovolné značení $\mathbf{M_0}$, tedy vrchol v grafu dosažitelnosti, které je dále považováno za výchozí. Tento algoritmus využívá datové struktury **X**, která je založena na zásobníkové datové struktuře k uložení zpracovávaných značení **M** a množiny všech značení $\mathbf{M_X}$ přímo dosažitelných ze značení **M**. Dále používá dvou množin značení, jednu množinu $\mathbf{X_{MA}}$ pro značení $\mathbf{M_A}$, ze kterých je dosažitelné dříve zvolené výchozí značení $\mathbf{M_0}$, a druhou množinu $\mathbf{X_{MN}}$ pro značení $\mathbf{M_N}$, ze kterých dříve zvolené výchozí značení $\mathbf{M_0}$ dosažitelné není. V prvním kroku algoritmu je zvolené výchozí značení $\mathbf{M_0}$ vloženo do množiny $\mathbf{X_{MA}}$ a dále je vloženo na vrchol prázdné struktury **X** společně se seznamem všech značení $\mathbf{M_X}$ přímo dosažitelných ze značení $\mathbf{M_0}$. Ve druhém kroku, pokud je struktura **X** prázdná algoritmus končí a množina $\mathbf{X_{MA}}$ obsahuje všechna značení, která jsou dosažitelná z výchozího značení $\mathbf{M_0}$. Pokud struktura **X** není prázdná, nahlédne se na její vrchol na seznam značení $\mathbf{M_X}$ přímo dosažitelných ze značení **M**, které je na vrcholu struktury **X**. Pokud je tento seznam prázdný bude se provádět backtracking s tím, že jsme dosáhli značení **M**, ze kterého je výchozí značení nedosažitelné. Pokud seznam není prázdný, pak je z něj vybráno libovolné značení $\mathbf{M_P}$ a pokračujeme třetím krokem. Ve třetím kroku mohou nastat čtyři možnosti. Pokud množina $\mathbf{X_{MA}}$ obsahuje značení $\mathbf{M_P}$, pak celá cesta, která je aktuálně projita a zachycena zásobníkem obsahuje značení, ze kterých je výchozí značení $\mathbf{M_0}$ dosažitelné. Všechna tato značení jsou přidána do množiny $\mathbf{X_{MA}}$, pokud se již v této množině nenacházejí, a bude se provádět backtracking. Jinou možností je, že množina $\mathbf{X_{MN}}$ obsahuje značení $\mathbf{M_P}$, pak musíme provádět backtracking s tím, že jsme dosáhli značení **M**, ze kterého je výchozí značení nedosažitelné. Další možností je že aktuálně projitá cesta vytvořila cyklus, tedy značení $\mathbf{M_P}$ je již na aktuální cestě jedou projito, pak musíme provádět backtracking s tím, že jsme dosáhli značení **M**, ze kterého je výchozí značení nedosažitelné. Pokud nenastala ani jedna z těchto možností, pak musíme pokračovat v procházení grafu do hloubky. Na vrchol struktury **X** umístíme značení $\mathbf{M_P}$ společně se seznamem všech značení $\mathbf{M_P}$ přímo dosažitelných ze značení $\mathbf{M_P}$ a pokračujeme druhým krokem. A nyní

k provádění samotného backtrackingu. Ten provádíme dokud není struktura **X** prázdná nebo dokud není na vrcholu struktury takové značení se seznamem přímo dosažitelných značení, který není prázdný. V každém kroku backtrackingu pak odstraníme z vrcholu struktury **X** značení **M** společně se seznamem přímo dosažitelných a pokud je backtracking prováděn z důvodu dosažení značení, ze kterého je výchozí značení $\mathbf{M_0}$ nedosažitelné, pak je značení **M** přidáno do množiny $\mathbf{X_{MN}}$, pokud se tam již nenachází.

4.4.3. Spolehlivost (soundness)

Spolehlivost již není vlastnost Petriho sítí, avšak je klíčovou vlastností Workflow sítí. Stejně jako Workflow sítě je tato vlastnost popsána v práci autorů Wil van der Aalsta a Kees van Heea *[7]*. Spolehlivost v podstatě odráží minimální podmínky kladené na validní specifikované workflow. Tedy, že workflow nesmí obsahovat žádné nepotřebné akce a každá instance takovéhoto workflow musí být vždy kompletně ukončena, což znamená, že neexistuje žádná akce nebo podmínka této instanci vlastní, která by byla aktivní i po ukončení workflow. Tedy všechny přechody **t** modelu workflow specifikovaného pomocí Workflow sítě musí existovat nějaké značení **M** dosažitelné z počátečního značení $\mathbf{M_0}$ takové, že je tomto značení **M** přechod **t** proveditelný, a navíc z každého tohoto značení **M** musí být dosažitelné koncové značení $\mathbf{M_E}$. A kompletního ukončení každé instance workflow znamená, že pro každou značku počátečního značení $\mathbf{M_0}$ na počátečním místě Workflow sítě je po konečném počtu provedení přechodů sítě dosaženo koncového značení $\mathbf{M_E}$, ve kterém je adekvátní značka pouze na koncovém místě Workflow sítě a všechna ostatní místa Workflow sítě jsou v tomto značení $\mathbf{M_E}$ prázdná.

Dle práce autorů Wil van der Aalsta a Kees van Heea lze problém spolehlivosti Workflow sítě převést na jiný problém, který vznikne vyjádřením vlastnosti spolehlivosti Workflow sítě pomocí vlastnosti živosti a omezenosti klasické Petriho sítě, která vznikne úpravou dané Workflow sítě. Výchozí Workflow síť je zkratována pomocí nového přechodu **t** spojujícího počáteční a koncové místo této sítě. Problém pak spočívá v ověření živosti a omezenosti této zkratované Workflow sítě, neboť platí, že Workflow síť **W** je spolehlivá právě tehdy, když je tato zkratovaná Workflow síť $\mathbf{W_{SC}}$ živá a omezená.

Algoritmus rozhodující zda je Workflow síť živá tedy plně vychází z představeného principu, kdy je v prvním kroku daná Workflow síť zkratována a následně jsou u ní pomocí dříve popsaných algoritmů ověřeny vlastnosti živosti a omezenosti.

4.5. Model Petriho sítě

Výše popsané základní prvky teorie Petriho sítí, jejich syntaktickou stavbu, sémantiku a následně algoritmy pro konstrukci grafu dosažitelnosti a algoritmy rozhodující o vlastnostech Petriho a Workflow sítí jsem postupně navrhnul a naimplementoval co nejobecněji tak, aby byl výsledný balíček použitelný i mimo tuto diplomovou práci. Opět jsem postupoval od návrhu dobře definovaných rozhraní, které vystihují problematiku Petriho sítí, k finální implementaci celého balíčku.

4.5.1. Návrh a implementace

V prvním kroku návrhu jsem navrhl rozhraní samotných základních prvků Petriho sítě. Na vrcholu hierarchie vrcholů Petriho sítě je rozhraní **IPetriNetNode**, které definuje společné metody pro místa a přechody Petriho sítě, které jsou následně popsány rozhraními **IPlace** a **ITransition** rozšiřujícími základné rozhraní **IPetriNetNode**. Tato rozhraní definují metody pro práci s příchozími a odchozími hranami vstupujícími a vystupujícími z těchto vrcholů. Samotné hrany pak jsou popsány superrozhraním **IArc**, které rozšiřují dvě rozhraní specializovaná na dva různé typy hran, které se v Petriho sítích nacházejí, a to **IPlaceTransitionArc** a **ITransitionPlaceArc**. V dalším kroku jsem definoval rozhraní specifikující metody pro práci se značením Petriho sítě **IMarking**. Toto rozhraní obsahuje větší množství definovaných metod, neboť právě práce se značením sítě je stěžejním úkolem většiny algoritmů a také samotná podstata značení Petriho sítě je složitá k zachycení. Následně jsem definoval rozhraní Petriho sítě **IPetriNet**, které definuje všechny veřejně dostupné metody pro práci s Petriho sítí a s Workflow sítí. Tedy metody pro přidávání, odebírání a získávání prvků modelu, metodu pro specifikaci počátečního značení sítě a metody, které jsou budou zodpovědné za rozhodování o vlastnostech omezenosti, živosti a spolehlivosti této sítě. Pro potřeby zpracování a konstrukce grafu dosažitelnosti jsem nakonec definoval rozhraní **IReachabilityGraph** reprezentující graf dosažitelnosti jako celek a následně rozhraní **IReachabilityGraphNode**, které zachycuje metody pro práci s vrcholy grafu dosažitelnosti a hran mezi nimi. Při návrhu tohoto rozhraní jsem využil myšlenky návrhového vzoru kompozit.

Po dokončení návrhu tohoto modelu jsem výše popsána rozhraní doplnil implementačními třídami, které tato rozhraní implementují. Implementace je obecná a nezávislá na dalších částech této práce. Opět jsem i zde využil algoritmu přidělování unikátních ID jednotlivým prvkům, který využívá aktuálních časových razítek, v čase instanciace objektu a následnému ověření zda takovéto ID se již nenachází v množině existujících vydaných unikátních ID, která je implementována jako statická proměnná třídy **PetriNetNode**. Tři nejobsáhlejší implementační třídy jsou pak třída **PetriNet** implementující rozhraní **IPetriNet**, která zapouzdřuje veškeré metody

popisující Petriho síť včetně algoritmu konstrukce grafu dosažitelnosti, dále třída **ReachabilityGraph** implementující rozhraní **IReachabilityGraph**, která zapouzdřuje veškeré procesy zpracování grafu dosažitelnosti včetně metod algoritmů rozhodujících právě na základě grafu dosažitelnosti o živosti a omezenosti Petriho sítě, a posledním velkým celkem je třída **Marking** implementující rozhraní **IMarking**, která obsahuje všechny metody pro práci se značením Petriho sítě. Navíc pro potřeby zpracování jsem v této třídě implementoval metodu **clone()** a přetížil standardní metodu **equals(Object)** a následně tak metodu **hashCode()**.

4.5.2. Implementace Algoritmu konstrukce grafu dosažitelnosti

Pro implementaci konstrukce grafu dosažitelnosti jsem zvolil dříve popsaný algoritmus, který pokračuje v konstruování grafu dosažitelnosti pouze ze značení, která jsou **n**-omezená pro předem zvolenou celočíselnou kladnou konstantu **n**. Tedy hledá další vývoj značení pouze z těch značení, která mají na každém místě Petriho sítě počet značek menší nebo roven této zvolené konstantě **n**.

Algoritmus je implementován ve třídě **PetriNet**, jejíž instance reprezentuje Petriho síť, která je vstupem tohoto algoritmu. Druhým vstupem algoritmu je celočíselná kladná konstanta **n**. Algoritmus pracuje se grafovou datovou strukturou **X**, která reprezentuje graf dosažitelnosti, implementačně s instancí třídy **ReachabilityGraph**. Každý vrchol této struktury tvoří značení **M** a hrany mezi dvěma značeními **M1** a $\mathbf{M_2}$ jsou tvořeny přechodem **t**, jehož provedením při značení $\mathbf{M_1}$ získáme značení $\mathbf{M_2}$. Implementačně je každý vrchol reprezentován instancí třídy **ReachabilityGraphNode**. Dále se používá množina již projitých značení $\mathbf{S_M}$ a frontou, **LinkedList**, značení označených k projití $\mathbf{Q_M}$. Na začátku je **X** prázdná, do $\mathbf{S_M}$ je přidáno počáteční značení Petriho sítě $\mathbf{M_0}$ a do $\mathbf{Q_M}$ je také vloženo $\mathbf{M_0}$. V prvním kroku se z fronty $\mathbf{Q_M}$ odebere první značení $\mathbf{M_P}$ označené k projití. Pokud je fronta $\mathbf{Q_M}$ prázdná algoritmus končí a struktura **X** reprezentuje graf dosažitelnosti. Jinak se pro každý přechod $\mathbf{t_P}$ proveditelný při značení $\mathbf{M_P}$ provede druhý krok. Ve druhém kroku se provede přechod $\mathbf{t_P}$ a získáme nové dosažené značení $\mathbf{M_N}$. Pokud množina $\mathbf{S_M}$ toto značení $\mathbf{M_N}$ neobsahuje, pak je toto značení přidáno do množiny $\mathbf{S_M}$. Dále pokud má nové značení $\mathbf{M_N}$ na každém místě sítě počet značek menší nebo roven **n**, pak je znační $\mathbf{M_N}$ přidáno také na konec fronty $\mathbf{Q_M}$. Dále je ve struktuře **X** vytvořen nový vrchol, který reprezentuje značení $\mathbf{M_N}$ a nová hrana, která vede ze značení $\mathbf{M_P}$ do značení $\mathbf{M_N}$ a reprezentuje přechod $\mathbf{t_P}$.

4.5.3. Implementace algoritmů ověřujících vlastnosti sítě

Algoritmy pro ověření omezenosti a živosti Petriho sítě jsou v tomto balíčku implementovány dle dříve popsaných algoritmů. Konkrétně pomocí jednoho průchodu výpočtu živosti na základě

grafu dosažitelnosti. Tedy pokud je počáteční značení dosažitelné ze všech možných značení stromu dosažitelnosti , pak je daná Petriho síť živá. Při tomto průchodu grafem dosažitelnosti je také u každého značení zkontrolována jeho omezenost vzhledem ke zvolené konstantě **n** a tedy na konci průchodu všemi dosažitelnými značeními je algoritmus schopen odpovědět, zda je daná Petriho síť **n**-omezená. Spolehlivost Workflow sítě se pak rozpadá na otázku, zda je zkratovaná Workflow síť živá a zároveň **n**-omezená. Všechny tyto algoritmy jsou implementovány ve třídě **ReachabilityGraph**.

5. Převod diagramu aktivit UML 2.0 na Workflow síť

Nyní, když jsem již nastínil, jakou podmnožinu prvků diagramu aktivit budu při modelování používat, sémantiku těchto prvků, a když jsem alespoň částečně popsal potřebnou část teorie Petriho sítí, prvků a vlastností Petriho sítě, a v neposlední řadě také nastínil pár detailů Workflow sítí a řešení problematiky spolehlivosti Workflow sítě, přichází na řadu otázka, jak tyto celky propojit dohromady tak, aby nám daly odpověď na otázku, jak formálně verifikovat model workflow specifikovaný pomocí diagramu aktivit UML 2.0 pomocí Workflow sítí.

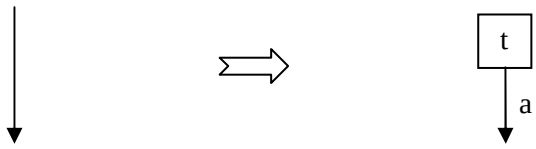
Jak tedy tyto celky propojit? Odpověď je nasnadě – převodem na základě jednoznačně definovaných pravidel. Využiji skutečnosti, že jsem již dříve specifikoval sémantiku jednotlivých prvků diagramu aktivit. Při tomto převodu se budeme snažit pomocí prvků Workflow sítě, respektive pomocí prvků Petriho sítě, vyjádřit sémantiku modelu diagramu aktivit. Tak vzniknou pravidla umožňující takovýto převod provést deterministicky a především automatizovaně.

5.1. Pravidla převodu

Tato pravidla se dají rozdělit podle jednotlivých prvků diagramu aktivit UML 2.0. Myšlenkou všech pravidel je zachytit sémantiku prvku diagramu aktivit a namapovat ji na určitý počet prvků Petriho sítě. Tyto prvky pak tvoří určité celky – stavební bloky nové Petriho sítě. Každý takový stavební blok má jedno či více určitých vstupních míst. Navržená pravidla umožní kompletní deterministický převod, jehož výsledkem bude Petriho síť, která však nebude minimální. V této práci se již nezabývám optimalizací a minimalizací vzniklé Petriho sítě.

5.1.1. Řídicí tok (Control flow)

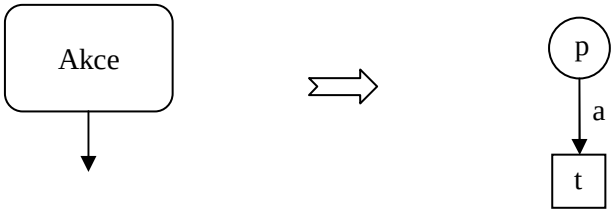
Řídící tok je při převodu mapován na přechod **t** Petriho sítě a hranu **a** z něj vedoucí do bloku reprezentující prvek diagramu aktivit.



Obrázek 9 – mapování řídícího toku na prvky Petriho sítě

5.1.2. Akce (Action node)

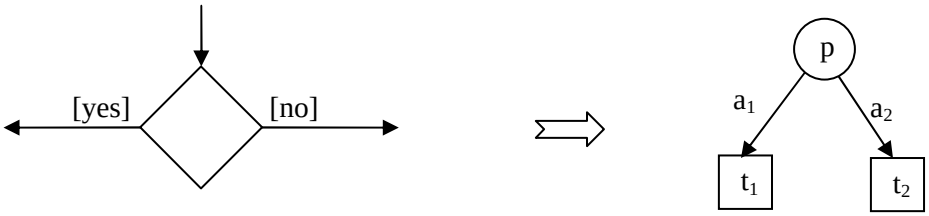
Akce je snadno mapovatelná na místo Petriho sítě. Pokud je akce spuštěna, může trvat určitý čas, než je akce dokončena. Po dokončení akce je okamžitě aktivován její odchozí řídící tok. To odpovídá místu Petriho sítě, ve kterém se nachází značka. Z teorie Petriho sítí vyplývá, že přechod propojený s takovým místem, ve kterém se nachází značka, je proveditelný, avšak k jeho provedení může dojít jak okamžitě, tak až za určitou dobu. Pokud z akce vystupuje řídící tok, pak je vytvořen celý stavební blok obsahující místo **p**, které je vstupním místem tohoto bloku, přechod **t** reprezentující tento řídící tok a hrana **a** od místa **p** k přechodu **t**.



Obrázek 10 – mapování akce na prvky Petriho sítě

5.1.3. Rozhodovací blok (Decision node)

Rozhodovací blok je také mapován na místo **p** Petriho sítě, které je vstupním místem tohoto bloku. Pro každý z něj vystupující řídící tok s podmínkou či bez je následně vytvořen samostatný přechod $\mathbf{t_i}$, který je spojen hranou $\mathbf{a_i}$ tento tok reprezentující, vedoucí z místa **p** do tohoto přechodu $\mathbf{t_i}$. Vzniká tak opět celý stavební blok.

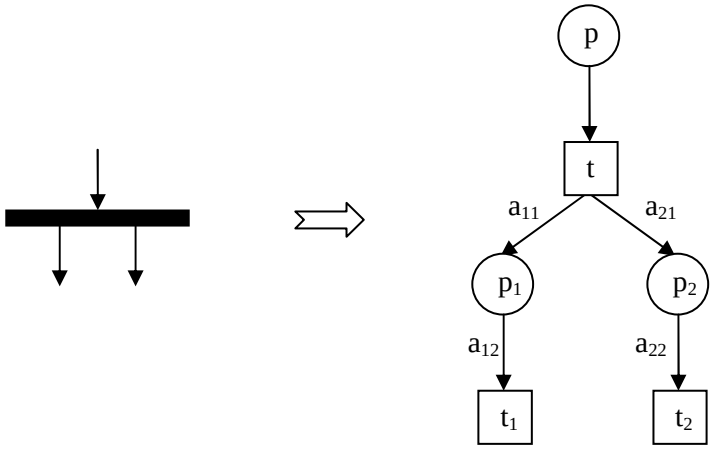


Obrázek 11 – mapování rozhodovacího bloku na prvky Petriho sítě

5.1.4. Rozdělovací synchronizační blok (Fork node)

Rozdělovací synchronizační blok zachycuje rozdělení workflow na konečný počet paralelně zpracovávaných větví. Tuto paralelitu je potřeba také vyjádřit pomocí prvků Petriho sítě. Proto je mapování částečně složitější. Pro rozdělovací synchronizační blok je nejprve vytvořeno místo **p**, které je vstupním místem tohoto bloku, přechod **t** a hrana **a** vedoucí z místa **p** do přechodu **t**. Pro každý řídící tok z množiny odchozích řídících toků je pak vytvořeno místo $\mathbf{p_i}$, které je propojeno hranou $\mathbf{a_{i1}}$ vedoucí od přechodu **t** do místa $\mathbf{p_i}$, a následně je vytvořen přechod $\mathbf{t_i}$ a hrana $\mathbf{a_{i2}}$

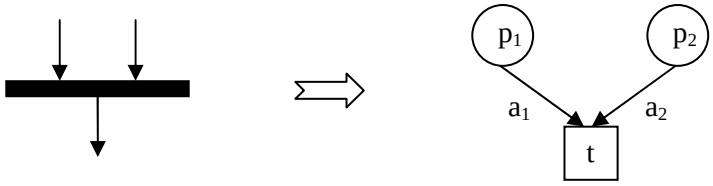
vedoucí od místa $\mathbf{p_i}$ do přechodu $\mathbf{t_i}$. Vznikne tak komplexní stavební blok z prvků Petriho sítě, který zachycuje rozdělení zpracování workflow do paralelních větví.



Obrázek 12 – mapování rozdělovacího synchronizačního bloku na prvky Petriho sítě

5.1.5. Slučovací synchronizační blok (Join node)

Slučovací synchronizační blok naopak zachycuje sloučení konečného počtu paralelně zpracovávaných větví workflow. Pro zachycení této podstaty jsem navrhl následující postup. Nejprve je vytvořen přechod **t** reprezentující aspekt spojení paralelních větví a poté je pro každý řídící tok z množiny příchozích řídících toků vytvořeno místo $\mathbf{p_i}$ reprezentující konec paralelní větve a hrana $\mathbf{a_i}$ vedoucí od místa $\mathbf{p_i}$ do přechodu **t**. Místa $\mathbf{p_i}$ jsou vstupními místy tohoto bloku.



Obrázek 13 – mapování slučovacího synchronizačního bloku na prvky Petriho sítě

5.1.6. Počáteční uzel (Activity initial node)

Počáteční uzel není potřeba vyjadřovat pomocí žádného prvku Petriho sítě. Počáteční uzel v podstatě pouze určuje místo počátku workflow. Tento fakt lze v Petriho síti zachytit pomocí počátečního značení $\mathbf{M_0}$ se značkou na příslušném místě. Ve Workflow síti je pak navíc potřeba označit příslušné místo jako počáteční místo sítě $\mathbf{p_0}$. Toto příslušné místo je vstupní místo stavebního bloku, který reprezentuje prvek diagramu aktivit, do kterého vede řídící tok z počátečního uzlu.

5.1.7. Koncový uzel (Activity final node)

Koncový uzel je snadno mapovatelný na místo **p** Petriho sítě. Pro potřeby Workflow sítě je však potřeba navíc označit toto místo za koncové místo sítě $\mathbf{p_{END}}$.

5.2. Implementace mapování

Základem implementace mapování se staly právě stavební bloky Petriho sítě reprezentující prvky modelu diagramu aktivit UML 2.0. Společné metody všech stavebních mapovacích bloků jsou specifikovány rozhraním **IMappingNode**, které pak třída každého stavebního bloku implementuje. Jedná se o metodu **connectToMappingNode(IMappingNode)**, která slouží ke spojení instance jednoho bloku na instanci bloku jiného a to tehdy, pokud je mezi prvky modelu diagramu aktivit, ze kterých tyto bloky vznikly, řídící tok. A pak o metodu **getIncomingArcPlace()**, která vrací vstupní místo instance mapovacího bloku, které bude použito k napojení instance bloku jiného. Každá implementační třída mapovacího bloku pak pomocí parametru v konstruktoru získá referenci na instanci objektu implementující rozhraní **IPetriNet**, reprezentující Petriho síť, do které pak bude přispívat všemi vytvořenými prvky tento blok reprezentující. Tyto prvky jsou v každé implementační třídě mapovacího bloku vytvořeny v privátní metodě **construct()** a to přesně podle definovaných pravidel převodu popsaných výše. Z určitého hlediska reprezentují tyto implementační třídy **ActionMappingNode**, **DecisionNodeMappingNode**, **ForkNodeMappingNode**, **JoinNodeMappingNode**, **ActivityInitialNodeMappingNode** a **ActivityFinalNodeMappingNode** určitým způsobem mechanismus návrhového vzoru fasáda, neboť zapouzdřují složitou logiku pomocí jednoduchého rozhraní.

Celý mechanismus a logiku mapovaní v několika krocích pak zastřešuje třída **ActivityDiagramToPetriNetMapper**. Instance této třídy pomocí několika metod nejdříve pro vstupní model diagramu aktivit reprezentovaný instancí třídy implementující IActivityDiagram zkonstruuje jednotlivé stavební bloky budoucí Petriho sítě, které následně propojí a podle potřeby takto vytvořený model Workflow sítě zkratuje.

5.3. Zpětná vazba

Jak sestrojit model diagramu aktivit je nám známo, cesta směrem od modelu diagramu aktivit směrem k modelu Workflow sítě je již popsána a jak takový model formálně verifikovat je již také objasněno. Ale jak naložit s výsledky formální verifikace modelu takto vytvořené Workflow sítě? Nejlepším řešením je zpětné promítnutí zanalyzovaných vlastností modelu Workflow sítě zpět do modelu diagramu aktivit. Právě pro tento účel jsem se rozhodl implementovat zpětnou

vazbu prvků modelu Workflow sítě tak, aby byly schopny udržovat referenci na prvek modelu diagramu aktivit, ze kterého pomocí mapovaní vznikly.

Z této myšlenky vzešlo rozhraní **IFeedbackPetriNetNode** specifikující metodu **getMappedActivityDiagramNode()**, a dvě konkrétní implementační třídy **FeedbackPlace** dědící ze třídy **Place** a **FeedbackTransition** dědící ze třídy **Transition**. Tyto třídy pouze rozšiřují standardní prvky Petriho sítě právě o schopnost zpětné vazby. Tyto třídy jsou pak využity v implementačních třídách mapovacích stavebních bloků namísto klasických tříd **Place** a **Transition**. Pokud tedy během formální verifikace modelu Workflow sítě narazíme na deadlock vznikající na určitém přechodu této sítě, jsme schopni zpětně označit prvek v modelu diagramu aktivit, který tento deadlock pravděpodobně způsobuje.

6. Nástroj pro editaci a verifikaci Workflow sítě

V kapitole třetí jsem popsal podmnožinu prvků diagramu aktivit jazyka UML 2.0 pro specifikaci workflow, jejich model a způsob implementace samostatného balíčku zapouzdřující tento model a jeho semiformální verifikaci. V kapitole čtvrté jsem popsal teorii Workflow sítí, její základní prvky, jejich model a způsob implementace samostatného balíčku zapouzdřující tento model a jeho formální verifikaci. A v kapitole páté možnosti převodu modelu diagramu aktivit jazyka UML 2.0 na model Workflow sítě a také implementaci tohoto převodu. Tyto připravené balíčky využiji právě v nástroji pro editaci a verifikaci Workflow sítí.

6.1. Vize nástroje

Nástroj bude umožňovat pohodlně graficky editovat workflow pomocí vybraných prvků diagramu aktivit jazyka UML 2.0, který bude automaticky na pozadí zachycovat workflow pomocí implementovaného modelu. Editor bude využívat grafickou paletu základních prvků diagramu aktivit jazyka UML 2.0 pro vytváření těchto prvků a dále nástroje pro výběr již vytvořených prvků, se kterými bude po výběru možno manipulovat, tedy přesouvat, mazat nebo přepojovat řídící toky. Tento nástroj bude umožňovat vytvořený model uložit do souboru pomocí specifikovaného XML formátu a posléze jej opět načíst. Dále bude nástroj schopen vytvořený model semiformálně verifikovat pomocí dříve implementovaného balíčku, následně jej převést na model Workflow sítě a formálně jej verifikovat a ověřit základní vlastnosti takto zkonstruované Workflow sítě. Výsledky verifikace a analýzy vlastností zkonstruované Workflow sítě bude nástroj schopen zobrazit a také promítnout zpět v grafické podobě do původního workflow.

Nástroj bude postaven jako plug-in platformy Eclipse a následně implementován jako samostatná aplikace postavená na běhovém prostředí Eclipse RCP, takže jej bude možno využít nejen v jakémkoliv IDE založeném na Eclipse verze 3.3 a výše, ale také jako samostatnou aplikaci. Grafický editor bude využívat rozšíření platformy Eclipse GEF a potažmo frameworku Eclipse Draw2D, které nabízí velmi široké možnosti vytváření grafických editorů pro platformu Eclipse. Ukládání a načítání modelu workflow bude zajištěno pomocí JAXB 2.0, jelikož tento ve verzi 2.0 a vyšší nabízí velmi pohodlné přizpůsobení marshallizace a demarshallizace objektů do a z XML souborů pomocí anotací tříd těchto objektů.

6.2. Použité technologie

V předchozí podkapitole jsem zmínil několik konkrétních frameworků a technologií, které si rozhodně zaslouží krátký obecný úvod a zevrubný popis, a to proto, že se jedná o technologie,

které nejsou masově rozšířeny, a tak o nich chybí obecné povědomí. Jedná se také o technologie nové, přinášející mnohá vylepšení k existujícím produktům či verzím. Tyto frameworky a technologie však nabízí velmi široké spektrum použití a silně demonstrují využití návrhových vzorů a také myšlenku škálovatelnosti známé základní platformy Eclipse.

6.3. Eclipse RCP

Eclipse platforma obecně slouží nejen jako základ pro nejrůznějších nástrojů a IDE, ale také jako framework vývojářům pro vytváření dalších nástrojů. Platforma Eclipse nabízí celou řadu frameworků a implementovaných služeb, nad kterými je možné snadno vyvíjet další plug-iny a jiná rozšíření pro existující nástroje, a především nabízí běhové prostředí, ve kterém tyto plug-iny načítány, integrovány a spouštěny. Důležitým aspektem je však architektura Eclipse platformy. Od Eclipse verze 3.0 je totiž architektura této platformy založena na zcela novém jádře, jehož základem je Eclipse RCP (Eclipse Rich Client Platform). Toto jádro je tvořeno množstvím komponent, které mohou být využity při implementaci jakéhokoliv klientského desktopového software, a které zajišťují jeho běh, UI, apod. A právě minimální podmnožina těchto komponent jádra platformy Eclipse, které zajišťují innicializaci, běh a UI platformy, tvoří Eclipse RCP. Důležité je také zmínit, že Eclipse RCP je open source. *[8]*

Základ platformy Eclipse RCP potřebný pro sestavení samostatné desktopové aplikace s UI nad touto platformou tvoří dva základní plug-iny **org.eclipse.ui** a **org.eclipse.core.runtime** a jejich nutné závislosti.

6.4. Eclipse GEF

Eclipse GEF (Graphical Editing Framework) umožňuje vývojářům vytvářet bohaté komplexní grafické editory za použití existujícího aplikačního modelu platformy Eclipse. Jedná se rovněž o open source projekt na kterém je založeno mnoho standardních nástrojů a editorů dodávaných v nejrůznějších distribucích platformy Eclipse jako například editor třídních diagramů, editor GUI pro AWT, editor GUI pro Swing a SWT apod. Pro vývoj těchto grafických editorů nabízí Eclipse GEF mnoho základních společných prvků, které mohou být využity přímo nebo snadno rozšířeny k dosažení požadovaných vlastností a funkcionality, tak aby odpovídaly potřebám konkrétního vyvíjeného editoru. Eclipse GEF je v podstatě složen ze dvou plug-inů. Základ tvoří samotný GEF plug-in **org.eclipse.gef**, který je závislý na druhém obsaženém plug-inu, kterým je konkrétně Draw2D plug-in **org.eclipse.draw2d** obsahující množství rozhraní, abstraktních i konkrétních tříd, jejichž primárním úkolem je poskytovat služby pro layout a rendering grafických prků. Draw2D plug-in podstatě zodpovídá za vizuální stránku GEF frameworku. Eclipse GEF při vývoji uživatelských grafických editorů ukázkově nutí vývojáře

využít architekturu implementující návrhový vzor MVC (model-view-controller). Tedy samotná architektura Eclipse GEF je postavena tak, že architekta i vývojáře vede k implementaci myšlenky MVC. Základem editoru je tedy datový model, který chceme upravovat v grafické podobě (pomocí grafického zobrazení), a prostředníkem mezi modelem a jeho grafickou podobou je samotný mechanismus editoru. *[9]*

Draw2D plug-in (**org.eclipse.draw2d**):

1. Zajišťuje služby pro efektivní layout prvků v rámci SWT Canvas a podporu pro jejich vykreslování
2. Obsahuje implementace různých grafických a geometrických tvarů (figures) a také implementace layout manažerů (layout managers)
3. Obsahuje implementace různých okrajů (borders)
4. Zajišťuje funkcionalitu a podporu grafických kurzorů (cursors) a nápověd (tooltips)
5. Zajišťuje podporu pro práci se spojovacími prvky a čarami, kotvení jejich konečných bodů na jiné prvky, jejich směrování a dekoraci
6. Umožňuje práci s několika průhlednými vrstvami (layers)
7. Umožňuje práci s flexibilním systémem souřadnic
8. Obsahuje implementaci funkcionality náhledu (overview window)
9. Zajišťuje podporu tisku a výstupu na tiskárnu

GEF plug-in (**org.eclipse.gef**):

1. Obsahuje implementace nástrojů pro výběr (selection tool), vytváření (creation tool), propojování (connection tool) a široký výběr (marquee tool) objektů v editoru
2. Obsahuje implementace několika palet (palettes) pro zobrazení a výběr těchto nástrojů
3. Zajišťuje funkce změny velikosti a kotvení propojovacích čar a prvků v editoru
4. Obsahuje implementace dvou typů pohledů Graphical view a Tree view
5. Obsahuje implementované jádro kontroleru, které zpracovává promítání změn mezi modelem a jeho view
6. Obsahuje třídy a rozhraní sloužící pro mapování interakcí na view v grafickém editoru na akce prováděné nad modelem
7. Podporu undo/redo právě díky využití akcí dle návrhového vzoru Command

6.5. JAXB 2.0

Java Architecture for XML Binding (JAXB) je technologie sloužící k rychlému vytváření vzájemných vazeb a propojení mezi XML schématy a Java třídami, které jsou využity pro zpracování XML dat v rámci Java aplikací. JAXB implementace definuje metody, pomocí kterých je možné načtení (unmarshalling) dat z instance XML dokumentu do stromu objektů tato data reprezentující, a pomocí kterých je možné provést opačný proces uložení (marshalling) dat ze stromu objektů do instance XML dokumentu. Prvotní verze JAXB 1.0 byla koncipována tak, že na základě XML schématu bylo nutné vygenerovat třídy, jejichž instance následně budou reprezentovat data zapouzdřená jednotlivými elementy v XML dokumentu, který je dle tohoto XML schématu validní. Tyto třídy poté mohly být využity v procesu marshallingu a unmarshallingu. Novější verze JAXB 2.0 nabízí mnohem pohodlnější přístup. Třídy, které jsou určeny k marshallingu a unmarshallingu postačí doplnit anotacemi, které jednoznačně definují způsob reprezentace dat v XML souboru. *[10]*

Zdůrazním zde rozdíl a výhody JAXB 2.0 oproti Simple API for XML (SAX) a Document Object Model (DOM). Přístup pomocí SAX umožňuje pouze sekvenční přístup k datům v XML dokumentu. Při tomto sekvenčním načítání je nutné data manuálně ukládat do vlastních objektů a vytvářet jejich strukturu k pozdějšímu použití. V podstatě je nutné implementovat logiku načítání XML dokumentu. Přístup pomocí DOM nabízí možnost načtení celého XML dokumentu do paměti v podobě stromu a poté jeho zpracování pomocí procházení tohoto stromu a ukládání dat do vlastních objektů a vytváření jejich struktury. Opět je nutné implementovat logiku načítání XML dokumentu. Pomocí JAXB 2.0 se pouze jedná o vytvoření a anotování tříd struktury dat. Logika načítání je plně v rukou implementace JAXB 2.0.

7. Eclispe GEF editor pro Workflow sítě

Prvním krokem při návrhu a implementaci nástroje bylo vytvoření plug-inu do platformy Eclipse. Jelikož se má jednat o grafický editor, zvolil jsem nejschůdnější cestu návrhu a implementaci s využitím dříve zmíněného frameworku Eclipse GEF. V podstatě tedy šlo o vytvoření běžného Eclipse plug-inu, který ovšem využívá balíku Eclipse GEF a Draw2D. Implementace takového grafického editoru postaveného nad Eclipse GEF se dá rozdělit do několika logických celků. Nejprve je potřeba vytvořit vrstvu, která bude v editoru sloužit jako model dat. Tento model by měl být na tomto frameworku nezávislý. Dále je potřeba vytvořit vizuální vrstvu editoru a prvky, které s ní bezprostředně souvisí, tedy implementace view. A posledním krokem je propojení těchto vrstev pomocí řídící logiky, tedy controlleru.

7.1. Datová vrstva editoru - model

Při návrhu a implementaci datového modelu editoru pro Workflow sítě je potřeba položit si otázku: „Co bude tento editor zachycovat a pomocí čeho?“ Odpovědí je diagram aktivit jazyka UML 2.0 reprezentující workflow a to pomocí množiny základních prvků definovaných v kapitole druhé. Datovým modelem tedy bude model diagramu aktivit UML 2.0. Nelze však přímo využít balíčku, který jsem dříve implementoval, nýbrž jeho rozšíření. Důvod je zřejmý. Datový model bude muset obsahovat nové rozšiřující informace, například o pozici jednotlivých prvku v diagramu. Navíc bude také potřeba naslouchat změnám, které z jakéhokoliv důvodu nastanou v samotném modelu a tyto změny promítnout ve vizualizaci tohoto modelu v editoru. V poslední řadě je to pak také myšlenka verifikace toho modelu a ta sebou nese nutnost v modelu zachytit, který případný prvek způsobil při verifikaci modelu problém.

Z těchto důvodů jsem navrhnul a následně naimplementoval balíček rozhraní rozšiřující příslušná existující základní rozhraní balíčku modelu diagramu aktivit UML 2.0. Stejně jako v něm je základním rozhraním **IActivityDiagramElementModel**, které rozšiřuje rozhraní **IActivityDiagramElement** a také rozhraní **IPropertySource**. Poslední zmíněné rozhraní je zde proto, aby bylo možné propagovat hodnoty dat modelu do Property view platformy Eclipse, což je také standardním postupem. Dalším rozhraním je **IActivityDiagramNodeModel**, které je výchozím rozhraním všech uzlů modelu a které dědí z **IActivityDiagramElementModel**. Toto rozhraní obsahuje metody get a set metody pro práci s pozicí prvku v editoru a také metody pro označení/zvýraznění uzlu. Následně je definována řada rozhraní pro jednotlivé uzly modelu, které nedeklarují žádné metody, ale pouze spojují rozhraní pocházející z balíčku modelu diagramu aktivit UML 2.0 s nově definovanými rozhraními. Tak například rozhraní **IActionNodeModel**, které reprezentuje uzel akce v modelu editoru, rozšiřuje rozhraní **IActionNode**

a rozhraní **IActivityDiagramNodeModel**. Následně jsou zde rozhraní **IControlFlowModel** a **IConditionalControlFlowModel**, která reprezentují řídící tok v modelu editoru, rozšiřující rozhraní **IControlFlow** a **IConditionalControlFlow** a také rozhraní **IActivityDiagramElementModel**. Celý diagram reprezentující workflow je reprezentován rozhraním **IActivityDiagramModel** rozšiřujícím rozhraní **IActivityDiagram** o deklarace metod pro práci s listenery změn modelu a zvýrazněním všech uzlů modelu.

Následně jsem tato rozhraní implementoval v implementačních třídách, které využívají myšlenky návrhového vzoru Facade, jelikož v implementačních metodách jsou volány metody implementačních tříd již existujícího balíčku modelu diagramu aktivit UML 2.0 a pak metody definované ve třídě zapouzdřující funkcionalitu propagátora dat a notifikátora změn dat modelu.

7.1.1. Perzistence

Pro perzistenci datového modelu jsem zvolil JAXB 2.0. Vytvořil jsem balík POJO tříd, reprezentujících jednotlivé prvky modelu, které jsou řádně anotovány dle specifikace JAXB 2.0 tak, aby je bylo možné pomocí marshallingu uložit do instance XML dokumentu nebo je z něj zpětně načíst pomocí unmarshallingu. K těmto třídám je možné pomocí JAXB 2.0 vygenerovat XML schéma popisující strukturu těchto XML dokumentů. Jedná se o třídy **ActionNodePersistent**, **DecisionNodePersistent**, **ForkNodePersistent**, **JoinNodePersistent**, **ActivityInitialNodePersistent**, **ActivityFinalNodePersistent**, **ControlFlowPersistent** a **ActivityDiagramPersistent**.

Základní myšlenkou perzistence tohoto modelu je uložení všech potřebných atributů. U všech jednotlivých uzlů jde o jejich pozici v diagramu a také jejich unikátní ID, která slouží pro identifikaci zdroje a cíle přiřazených řídících toků a také spárovaných rozdělovacích a slučovacích synchronizačních bloků. Prostředníkem mezi perzistentním modelem a modelem požitým v samotném editoru je třída **DiagramConstructor**, která umí provádět oboustranný převod mezi těmito modely.

7.2. Vizuální vrstva editoru - view

Vizuální vrstva je tvořena grafickými prvky, které mají za úkol vizuálně reprezentovat prvky modelu a zároveň zajišťovat funkcionalitu kotvení spojovacích čar v daných místech grafických prvků.

7.2.1. IFigures

Pro každý uzel jsem vytvořil a implementoval třídu reprezentující jeho grafickou vizualizaci s využitím tříd balíku Eclipse Draw2D tak, aby maximálně odpovídala specifikaci UML 2.0. Pro

rozdělovací a slučovací synchronizační blok jsem však musel k vizualizaci přidat rozlišovací prvek, kterým je malý šedý nápis fork potažmo join.

Obrázek 14 – uzly diagramu aktivit UML 2.0 vykreslené pomocí Draw2D

7.2.2. Anchors

Každý ze zmíněných uzlů musí mít také definované body pro kotvení spojovacích čar, v případě diagramu aktivit UML 2.0 řídících toků. Pro uzel akce, počáteční uzel a koncový uzel je snadné použít již existující implementace **ChopboxAnchor**, která vrací za kotevní bod ten, který leží na průsečíku spojovací čáry a prvku samotného. Pro rozhodovací blok jsem implementoval **DecisionNodeAnchor**, který vrací za kotevní bod roh prvku nejblíže ke spojovací čáře. A následně pro rozdělovací a slučovací synchronizační blok jsem implementoval **ForkNodeAnchor** a **JoinNodeAnchor**, které vracejí kotevní body nejblíže středu tohoto prvku.

7.3. Řídící logika editoru - controller

Nejrozsáhlejší částí celého editoru je pak samotná funkční a řídící logika editoru. Dá se také rozdělit do několika logických celků, z nich nejzajímavějším je asi množina tříd, které sehrávají roli konkrétních controllerů a propojují model dat s jeho view. Další dílčími částmi jsou například třída továrny, která se stará o instanciaci objektů modelu, balík tříd dle návrhového vzoru command, které mají za úkol provádět jednotlivé akce nad modelem dat, balík tříd různorodých politik editoru apod.

7.3.1. EditParts

Při implementaci funkcionality editoru jsem začal nejprve s implementací jednotlivých controllerů, neboli EditParts dle terminologie Eclipse GEF frameworku. Myšlenka frameworku je taková, že pro každý prvek modelu a jeho grafickou reprezentaci view existuje jeden EditPart, který tvoří jejich prostředníka. Každý editor má specifikovánu třídu továrny implementující rozhraní **EditPartFactory**, která je zodpovědná za vyváření instancí EditParts, tak jak je potřeba. Nejprve na začátku životního cyklu editoru je mu předána instance modelu, který je jeho vstupem, je využita právě tato továrna ke konstrukci jednotlivých EditParts pro jednotlivé prvky modelu. Po té je již každý EditPart zodpovědný za svůj prvek modelu, který spravuje, jeho případné potomky a spojovací čáry a také za vizuální prvek tento model reprezentující. Každý zásah do modelu zobrazeného v editoru je pomocí EditPart příslušného prvku promítnut do jeho

modelu a naopak je každý zásah do modelu pomocí EditPart promítnut do jeho vizuální podoby v editoru.

Eclipse GEF framework definuje řadu abstraktních tříd, ze kterých mohou jednotlivé EditParts dědit. Já jsem v implementaci využil abstraktní třídy **AbstractGraphicalEditPart**, která je určena pro prvky grafických editorů. Další zajímavá myšlenka, kterou jsem při návrhu jednotlivých EditParts využil vychází z faktu, že každý takový EditPart často pracuje s objektem modelu a tak nastává nutnost objekt modelu často přetypovávat. Využil jsem tedy možností programovacího jazyka a nejprve vytvořil abstraktní třídu **AbstractActivityDiagramNodePart**, která je zároveň generickým typem a definuje metodu **getCastedModel()**, která vrací objekt modelu přetypovaný dle specifikovaného parametrizovaného typu. Tato třída je využita jako supertřída pro všechny EditParts uzlů diagramu aktivit UML 2.0, které implementačně specifikují konkrétní parametrizovaný typ použitého modelu. Jedná se o třídy **ActionNodePart**, **DecisionNodePart**, **ForkNodePart**, **JoinNodePart**, **ActivityInitialNodePart** a **ActivityFinalNodePart**. Dalšími potřebnými EditParts jsou **ControlFlowPart** a **ConditionalControlFlowPart**, které dědí z **AbstractConnectionEditPart** a tvoří propojení mezi modelem řídících toků a funkcionalitou spojovacích čar grafického editoru GEF Frameworku. Posledním nezbytným controllerem je EditPart zapouzdřující logiku diagramu aktivit UML 2.0 jako celku – třída **ActivityDiagramPart**

7.3.2. EditPolicies a Commands

Politiky (EditPolicies) slouží k rozšíření základního chování EditParts, tedy role controlleru. EditPart může za pomocí zásuvného mechanizmu registrovat určité EditPolicies, které budou zodpovědné za další role, ve kterých se může daný EditPart vyskytovat. Pokud uživatel provede v editoru určitou akci nad vizualizací prvku, pak EditPart tohoto prvku převezme požadavek na provedení této akce. EditPart však tento požadavek neobsluhuje přímo, nýbrž prostřednictvím jednotlivých registrovaných EditPolicies, které jsou zodpovědné za vytvoření instance objektu reprezentuje Command, jehož spuštěním dojde k vykonání požadavku.

V rámci návrhu a implementace editoru workflow, který bude využívat umísťování prvků pomocí XY souřadnicového systému jsem musel vytvořit EditPolicy **WorkflowUMLDiagramXYLayoutEditPolicy**, která je registrována v EditPartu **ActivityDiagramPart** v roli Layout a zajišťuje vytváření objektů Command zodpovědných za vytváření nových uzlů diagramu aktivit v editoru a jejich posouvání/přemísťování. Další obecnou funkcionalitu odstraňování prvku z modelu zapouzdřuje EditPolicy **WorkflowUMLDiagramNodeComponentEditPolicy**, která je registrována v EditPartu každého uzlu v roli Component. Díky této EditPolicy je možné reagovat na požadavky ke smazání takového prvku. Další dvě EditPolicy zastřešují zodpovědnost za vytváření objektů Command, které vytvářejí či editují spojovací čáry reprezentující řídící toky, jde o třídy **ControlFlowEditPolicy** a

ControlFlowDeletePolicy. Dále jsem implementoval celou řadu tříd reprezentujících jednotlivé Commands zodpovědné za vykonání změn na modelu. Tyto třídy dědí ze třídy **Command** a díky myšlenkám návrhového vzoru Command umožňují vykonávání akcí, jejich undo a redo nebo např. rozhodování zda je taková akce nad daným modelem přípustná.

7.3.3. GraphicalEditor

Poslední částí jsou třídy samotného editoru. Jedná se o třídu **WorkflowUMLEditor** a třídu **WorkflowUMLEditorPaletteFactory**. Druhá zmiňovaná třída slouží jako továrna pro prvky palety a palety editoru samotné. Úkolem této třídy je tedy pouze vytvořit paletu a svázat její prvky s akcemi, které budou nad modelem vyvolány.

Třída **WorkflowUMLEditor** reprezentuje prvek samotného grafického editoru a dědí z abstraktní třídy GEF frameworku **GraphicalEditorWithPalette**. V této třídě jsou implementovány metody sloužící k nastavení vlastností editoru, pro konfiguraci grafického Viewera, pro nastavení továrny prvků, pro načtení a uložení modelu apod. Implementace je velmi jednoduchá, neboť plně vychází z potřeb definovaných v dokumentaci GEF frameworku.

Po dokončení implementace grafického editoru, vytvoření několika ikon a po následné implementaci tříd a vytvoření XML konfigurace nutné k tomu, aby byl tento editor využitelný jako plug-in platformy Eclipse, bylo již možné editor vyzkoušet přímo v použitém Eclipse IDE. Obecný postup vytváření a konfigurace plug-inů pro platformu Eclipse je nad rámec této práce a proto jej zde nebudu popisovat. Všechny tyto informace týkající se plug-inů je možné najít zdokumentované na stránkách platformy Eclipse. Osobně se v práci dále zaměřím na části či konkrétní třídy specifické pro workflow editor.

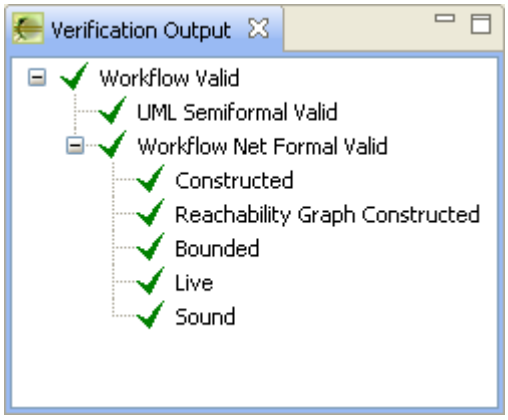
7.4. Další příspěvky plug-inu do Eclipse platformy

7.4.1. Verifikační View

Editor je již možné používat k editaci workflow pomocí diagramu aktivit UML 2.0, avšak chybí možnost jej v platformě Eclipse semiformálně a následně formálně verifikovat a výsledek zobrazit v rozumné podobě. K zobrazení výsledků procesu verifikací jsem se rozhodl k implementaci rozšiřujícího View pro platformu Eclipse postaveného na stromové struktuře. Při vytváření View je potřeba vytvořit třídu, která reprezentuje View samotné a buďto dědí ze třídy **ViewPart** nebo implementuje rozhraní **IViewPart**, dále balík tříd reprezentující model, který bude toto View zobrazovat, a ve finále Providers, tedy implementaci třídy **LabelProvider**, která je zodpovědná za vizualizaci prvků modelu ve View, a implementaci rozhraní **ITreeContentProvider**, které je zodpovědné za parsování struktury dat modelu.

Konkrétní třídou verifikačního View je **VerificationOutputView** a její implementace je triviální. Třída obsahuje pouze reference na použitý Viewer, konkrétně **TreeViewer**, na LabelProvider a na kořen stromu modelu dat, který bude pomocí Vieweru zobrazen, a dále základní metody pro vytvoření GUI View a nastavení jeho vstupu.

Použitý datový model vychází z požadavku zobrazovat výsledek jednotlivých akcí semiformální a formální verifikace, který může nabývat hodnot validní, nevalidní, nekontrolováno. K tomuto účelu jsem navrhnul výčtový typ **Validity** s těmito třemi možnými hodnotami. Následně jsem navrhnul a implementoval třídu **ValidationDataNode**, která je předkem všech uzlů verifikačního stromu. Každý uzel verifikačního stromu, který není listem, pak umí rozhodnout na základě svých potomků o hodnotě výsledku verifikace své části. Na obrázku 15 je vidět jednotlivé kroky verifikace, které ve výsledku vytvářejí jednoduchý strom.



Obrázek 15 – výsledné verifikační View po úspěšné verifikaci modelovaného workflow

7.4.2. Actions

Verifikační View je již funkční a je nyní potřeba vytvořit spouštěcí mechanismus verifikace modelu editoru. K tomu slouží další rozšíření platformy Eclipse - Action. Implementací Action v plug-inu je možné obohatit platformu o další akce napojená na tlačítka či položky menu. Pro potřeby spuštění procesu verifikace jsem implementoval třídy **WorkflowVerificationAction** a **WorkflowVerificationRetargetAction**. První jmenovaná třída dědí ze třídy **WorkbenchPartAction** a zapouzdřuje funkcionalitu spuštění procesu verifikace na modelu diagramu aktivit pomocí metod, které jsou implementovány v balíčku modelu dat diagramu aktivit UML 2.0, a následně převedení modelu do Workflow sítě a její formální verifikaci pomocí metod implementovaných v balíčku modelu Workflow sítě. Výsledek je převeden na stromovou strukturu modelu použitelného ve verifikačním View a také vizuálně překlopen do vizualizace samotného workflow v grafickém editoru. Třída **WorkflowVerificationRetargetAction** dědí ze třídy

RetargetAction a slouží k přemostění volání dříve specifikované akce z prostředí platformy Eclipse.

7.4.3. Wizard wfl diagramů

Dalším důležitým příspěvkem do Eclipse platformy je Wizard pro vytváření nových diagramů, které jsou spravovány pomocí Eclispe Workspace jako zdrojové soubory s příponou wfl. Tento Wizard umožňuje přidat nový zdrojový soubor do Workspace standardním způsobem pomocí dialogu „New“ stejně jako jakékoliv jiné zdrojové soubory. Wizard je velmi jednoduchý a skládá se pouze ze dvou implementačních tříd. Instance třídy **NewWorkflowDiagramWizard** reprezentuje objekt Wizarda a umožňuje jeho zapojení do platformy a třída **NewWorkflowDiagramWizardPage** zapouzdřuje samotnou stránku s nabídkou vytvoření zdrojového souboru a také implicitní obsah takového souboru.

7.4.4. Přítomnost ovládacích prvků

Aby bylo možné definovaných Actions v plug-inu využít v platformě Eclipse potřeba je s prostředím svázat pomocí třídy Contributoru. Proto vzniká implementace třídy **WorkflowUMLEditorActionBarContributor**, která má za úkol přispívat do hlavního menu položkou „Workflow“ a v ní vnořenou položkou „Verify Workflow“, která spouští akci verifikace modelu workflow v editoru, a dále do lišty nástrojů tlačítkem spouštějícím stejnou akci. Tyto položky se na svých místech objeví pouze tehdy, je-li při práci s Workbench platformy Eclipse otevřen Workflow editor a tak se také stanou použitelnými.

8. Eclipse RCP aplikace pro workflow nástroj

Posledním krokem ve vývoji tohoto nástroje je překlopení hotového Eclipse plug-inu se všemi rozšířeními do podoby plnohodnotné samostatné aplikace s využitím Eclipse RCP. Samostatný plug-in jehož součástí je grafický editor workflow, verifikační View, Wizard pro vytváření nových wfl diagramů a Actions propagované pomocí Cotributoru do menu a lišty nástrojů je vytvořen tak, že je použitelný v jakémkoli instanci platformy Eclipse verze 3.4 a výše a díky tomu je využitelný i v jakékoliv aplikaci postavené na Eclipse RCP 3.4 a výše.

8.1. Jádro RCP aplikace

Spouštěcí mechanismus celé aplikace tvoří třída **WorkflowUMLToolApplication**, která implementuje rozhraní **IApplication** a její 2 dvě metody **start()** a **stop()**, zodpovědné za zahájení a ukončení běhu RCP aplikace. Během zahájení běhu metoda **start()** vytváří instanci Workbench reprezentující celé pracovní prostředí platformy Eclipse a předává mu instanci objektu **WorkbenchAdvisor.**

8.1.1. Advisors

Konrétní implementací **WorkbenchAdvisor** v této aplikaci je třída **ApplicationWorkbenchAdvisor**. Tato třída je první ze tří tříd typu Advisor, které konfigurují různé aspekty a vlastnosti instance Workbench, a nastavuje výchozí Perspective, která má být použita po startu aplikace a také vytváří instanci třídy **WorkbenchWindowAdvisor** konkrétně **ApplicationWorkbenchWindowAdvisor**. **WorkbenchWindowAdvisor** zodpovídá a řídí stavový řádek, panel nástrojů, nebo velikost oken a jiných vlastností spojených s grafickými prvky platformy a také zodpovídá za vytvoření instance posledního Advisora, kterým je **ApplicationActionBarAdvisor**. Posledně jmenovaná třída slouží ke správě všech prvků, které spadají do skupiny ActionBar – tedy menu, kontextová menu, panely nástrojů a stavový řádek. Jejím hlavním úkolem je vytváření instancí Actions a jejich registrací a umístěním do jednotlivých menu či panelů nástrojů.

8.1.2. Perspective

Jak jsem již zmínil, třída **ApplicationWorkbenchAdvisor** má mezi jinými za úkol nastavit výchozí Perspective. Implementoval jsem proto třídu **WorkflowUMLToolPerspective**, která definuje přijatelnou Perspective pro tuto aplikaci rozdělenou na levý panel s navigačním View, střední panel pro editor a levý panel, ve kterém je v horní části standardní PropertiesView a v dolní části verifikační View.

8.2. Použití plug-inu v RCP aplikaci

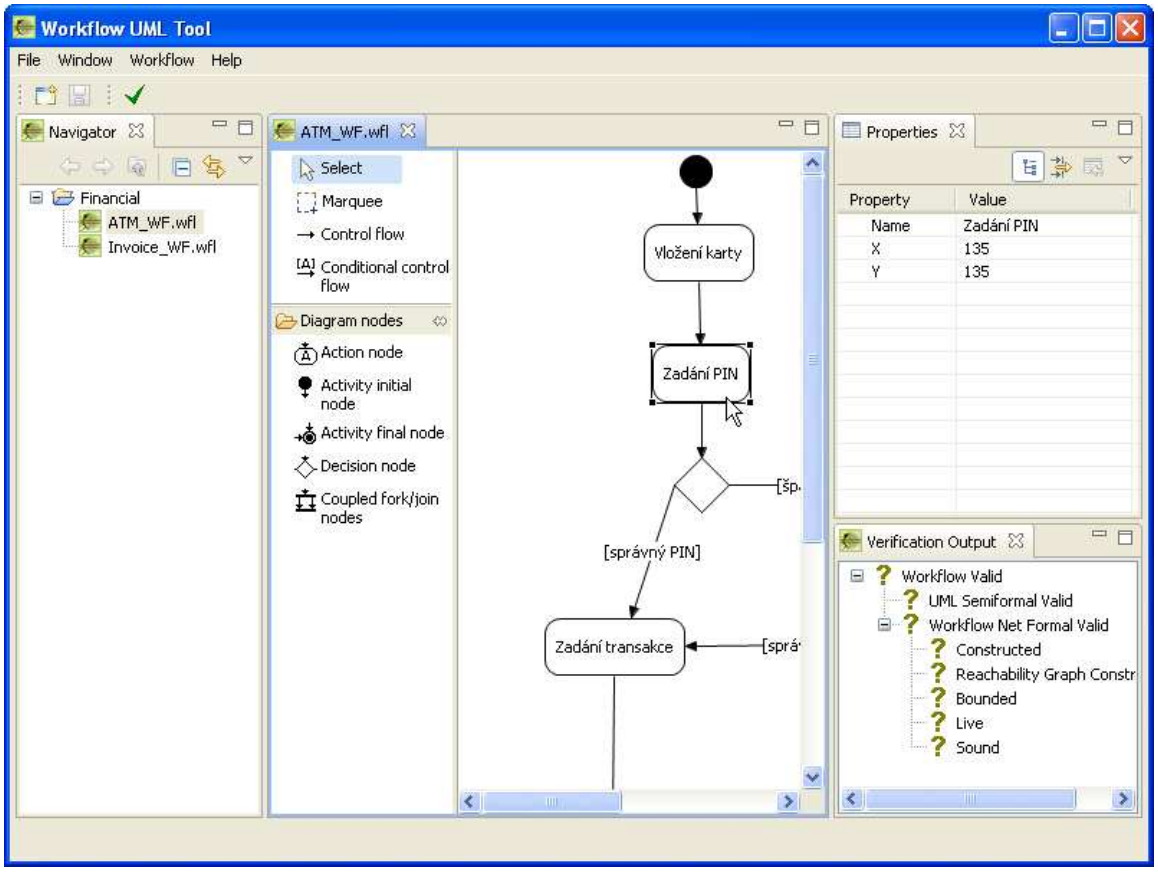
Konfigurace hotového plug-inu do aplikace je snadná a vychází z definice samotného plug-inu v souboru plugin.xml, který je standardní konfigurační soubor pro plug-iny Eclipse platformy. Jedná se o zahrnutí jednotlivých rozšíření hotového plug-inu do konfiguračního souboru RCP aplikace.

8.3. Doplňující části

Využití prázdné RCP aplikace sebou ale nese jedno úskalí. Bez dalších rozšíření není možné využít zprávy zdrojových souborů ve workspace, čímž dojde ke znemožnění správy a práce s workflow diagramy. Aby k tomu tedy nedošlo, je potřeba aplikaci rozšířit o závislosti platformy Eclipse, které zastřešují tuto funkcionalitu a naimplementovat jednoduchý **ProjectNavigatorView**, který slouží stejně jako klasický Project navigator známý z Eclipse.

8.4. Běh samostatné RCP aplikace

Výsledkem kompletní implementace je samostatná aplikace, která ke svému běhu potřebuje pouze Java JRE verze 1.5. Pro to, aby mohla tato aplikace implementující nástroj pro verifikaci Workflow sítě – Workflow UML Tool pracovat samostatně, je potřeba z ní vytvořit Eclipse RCP produkt. Eclipse IDE k tomu nabízí velmi šikovný export wizard, který dokáže celou Eclipse RPC aplikaci včetně všech nutných plug-inů a jiných závislostí elegantně zabalit do produktové podoby. Takto exportovaná aplikace se chová jako standardní Eclipse IDE a je spouštěna pomocí spustitelného souboru dané platformy, např. exe souboru pro platformu win32.



Obrázek 16 – ukázka Workflow UML Tool aplikace

9. Závěr

Z vlastních praktických zkušeností nabytých během více než 3-leté práce ve společnosti Tieto mohu s jistotou konstatovat, že podnikové procesy jsou velmi často skloňovaným pojmem. Ve společnosti takového formátu se s podnikovými procesy setkávají zaměstnanci každodenně při jakékoliv vykonávané pracovní činnosti. Nalezení, pojmenování, popsání, udržování a správa podnikových procesů je pro takovou společnost životně důležité. Při definici těchto procesů je třeba dbát na to, aby byly tyto definovány formálně správně a také aby byly validní. Všechna tato fakta mě vedla k výběru tohoto tématu diplomové práce. Získal jsem možnost rozšířit si znalosti spjaté s podnikovými procesy a workflow, nejen teoreticky, ale i prakticky se seznámit s problematikou a úskalími verifikace workflow. Konkrétně problematika formální verifikace workflow definovaných pomocí diagramů aktivit UML 2.0 byla pro mne velkým přínosem. Z teoretického hlediska jsem získal široké povědomí o tom, jaká úskalí tato problematika skrývá a také jaké jsou možnosti a způsoby vypořádání se s verifikací neformálně definovaných modelů definovaných diagramem aktivit UML 2.0.

Od února roku 2008, kdy jsem začal zpracovávat tuto diplomovou práci, jsem již přečetl mnoho elektronických článků, webových konferencí, odborných publikací, které mi pomohly proniknout do problematiky verifikace workflow a získat tak nad touto tématikou částečný nadhled. Toto téma je celosvětově probírané na mnoha konferencích a z teoretického hlediska je stále otevřené, neboť pro formální teoretické ověření správnosti definovaného workflow se využívá některých složitých algoritmů, které často narážejí na problém s exponenciální složitostí. Proto se v této oblasti hledají optimalizované postupy a stále se otevřeně diskutuje a dokazuje, zda je tento problém obecně řešitelný s rozumnou, tedy alespoň polynomiální složitostí.

V rámci této práce jsem úspěšně navrhnul a naimplementoval obecný znovupoužitelný balíček provázaných rozhraní a také implementačních tříd pro práci s diagramem aktivit UML 2.0 a následně také balíček rozhraní a tříd pro práci s P/T Petriho sítěmi a také Workflow sítěmi. Oba tyto balíčky jsou naprosto nezávislé na předkládaném řešení a mohou být dále využity včetně algoritmů v implementačních třídách. Nad těmito dvěma zmíněnými balíčky jsem naimplementoval funkční jádro nástroje, které dokáže převádět model tvořený diagramem aktivit na workflow model reprezentovaný Petriho/Workflow sítí. Toto jádro je také schopné načítat/ukládat model tvořený diagramem aktivit z/do souboru v XML formátu. Nad tímto nezávislým jádrem jsem vytvořil adaptační mezivrstvu, nad kterou jsem pomocí Eclipse GEF frameworku vybudoval grafickou platformu nástroje pro editaci a verifikaci workflow kreslených pomocí diagramů aktivit UML 2.0. V poslední fázi jsem tento nástroj převedl do formy, ve které

je využitelný nejen jako plug-in do jakékoliv Eclipse platformy, ale také jako samostatná aplikace implementovaná nad platformou Eclipse RCP.

Vlastním přínosem této práce je nástin semiformální verifikace workflow specifikovaných pomocí podmnožiny prvků digramu aktivit UML 2.0, který lze dále rozšířit o jiná, další pravidla a sémantická omezení tohoto modelu. Také v této práci uvádím postup, jak model workflow specifikovaný pomocí prvků digramu aktivit UML 2.0 mapovat na prvky modelu Petriho sítě, potažmo Workflow sítě, které lze verifikovat formálně na základě všeobecně známých metod a z jejichž vlastností můžeme zpětně zkoumat vlastnosti původního modelu. Nesporným přínosem je také prezentovaný náhled na moderní technologie a frameworky jako je JAXB 2.0, Eclipse RCP a Eclipse GEF, které lze cíleně využít při tvorbě Java aplikací s GUI nebo konkrétněji při tvorbě grafických editorů.

Z pohledu dalšího možného vývoje projektu jako takového se zde naskýtají možnosti rozšíření stávající projekt o export generované mapované Workflow sítě do PNML (Petri Net Markup Language) formátu založeném na XML, který by umožnil její verifikaci a ověření jejích vlastností v jiných sofistikovaných nástrojích. Další možností rozšíření stávajícího projektu je například také vizualizace generované mapované Workflow sítě. Dále, jak jsem již zmínil dříve, je možné rozšířit pravidla semiformální verifikace modelu workflow specifikovaných pomocí podmnožiny prvků digramu aktivit UML 2.0. Příkladem by mohla být kontrola a analýza podmínek řídících toků vycházejících z rozhodovacího bloku.

10. Literatura

[1] VONDRÁK, Ivo. Metody byznys modelování : pro kombinované a distanční studium.: 2004. 92 s. Dostupný z WWW: http://vondrak.cs.vsb.cz/download/Metody_byznys_modelovani.pdf

[2] ESHUIS, Rik. Semantics and Verification of UML Activity Diagrams for Workflow Modelling.: 2002. 240 s. Dizertační práce. ISBN 90-365-1820-2.

[3] UML 2.0 Superstructure specification [online]. 2005 [cit. 2009-01-17]. Dostupný z WWW: http://www.omg.org/spec/UML/2.0/

[4] OMG, UML 2.0 Superstructure specification: 2005. 710 s. Dostupný z WWW: http://www.omg.org/spec/UML/2.0/Superstructure/PDF/

[5] MARKL, Jaroslav. Učební texty k předmětu Petriho sítě I.: 2006. Dostupný z WWW: http://www.cs.vsb.cz/markl/pn/

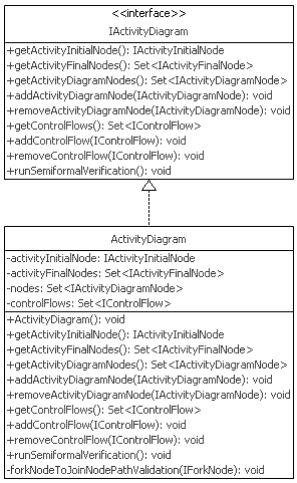
[6] Petri Nets World: Online Services for the International Petri Nets Community [online]. [cit. 2009-02-18] . Dostupný z WWW: http://www.informatik.uni-hamburg.de/TGI/PetriNets/

[7] VAN DER AALST, Wil, VAN HEE, Kees. Worklflow Management, Models, Methods, and Systems: MIT Press, 2002. 384 s. ISBN 978-0-262-01189-1

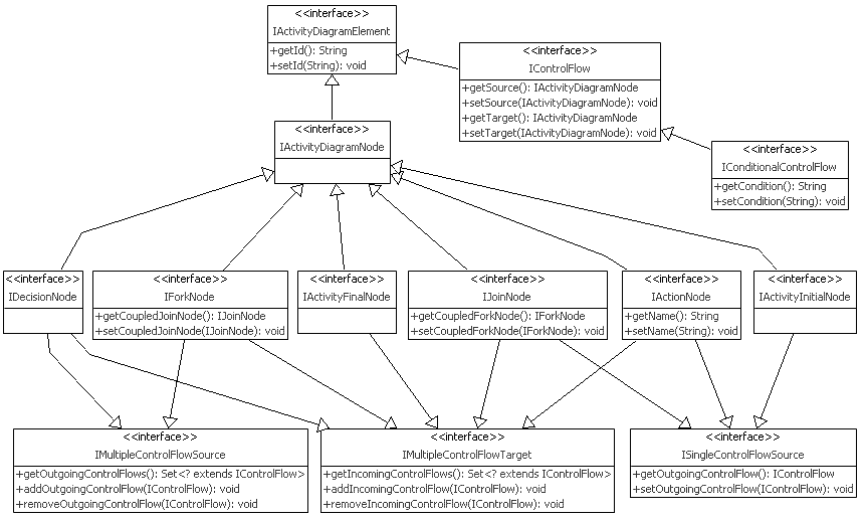
[8] Rich Client Platform – Eclipsepedia [online]. [cit. 2009-03-11] . Dostupný z WWW: http://wiki.eclipse.org/index.php/Rich_Client_Platform/

[9] Eclipse Graphical Editing Framework (GEF) [online]. [cit. 2009-03-12] . Dostupný z WWW: http://www.eclipse.org/gef/

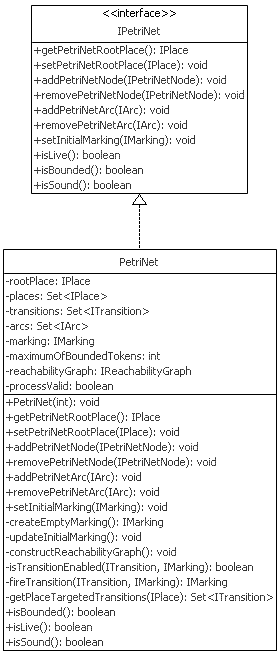
[10] jaxb: JAXB Reference Implementation [online]. [cit. 2009-03-11] . Dostupný z WWW: https://jaxb.dev.java.net/



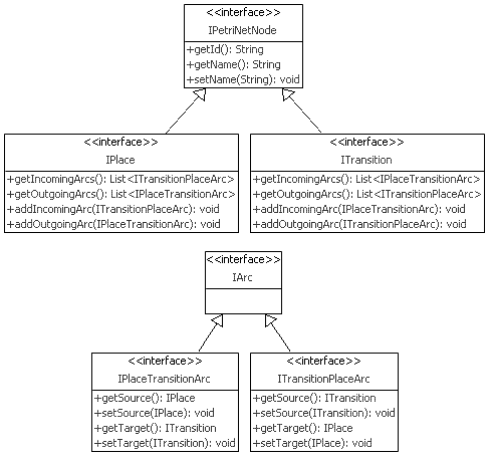
Příloha 1 – Class diagram: UML 2.0 diagram interfaces



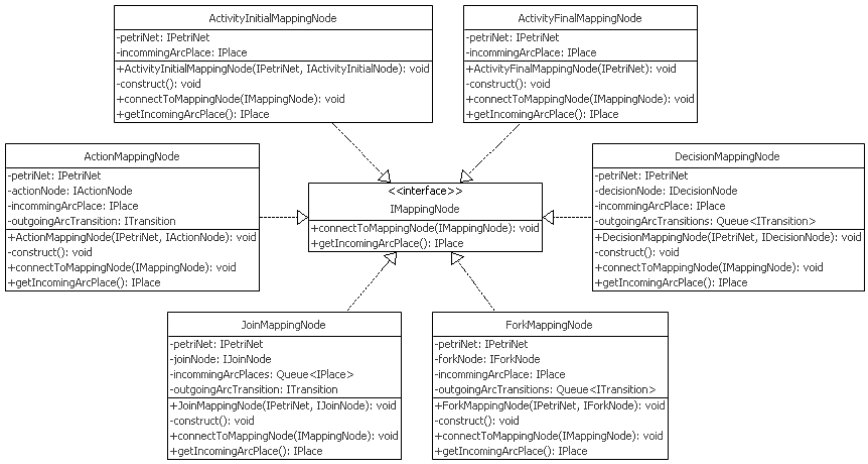
Příloha 2 – Class diagram: UML 2.0 diagram elements interfaces



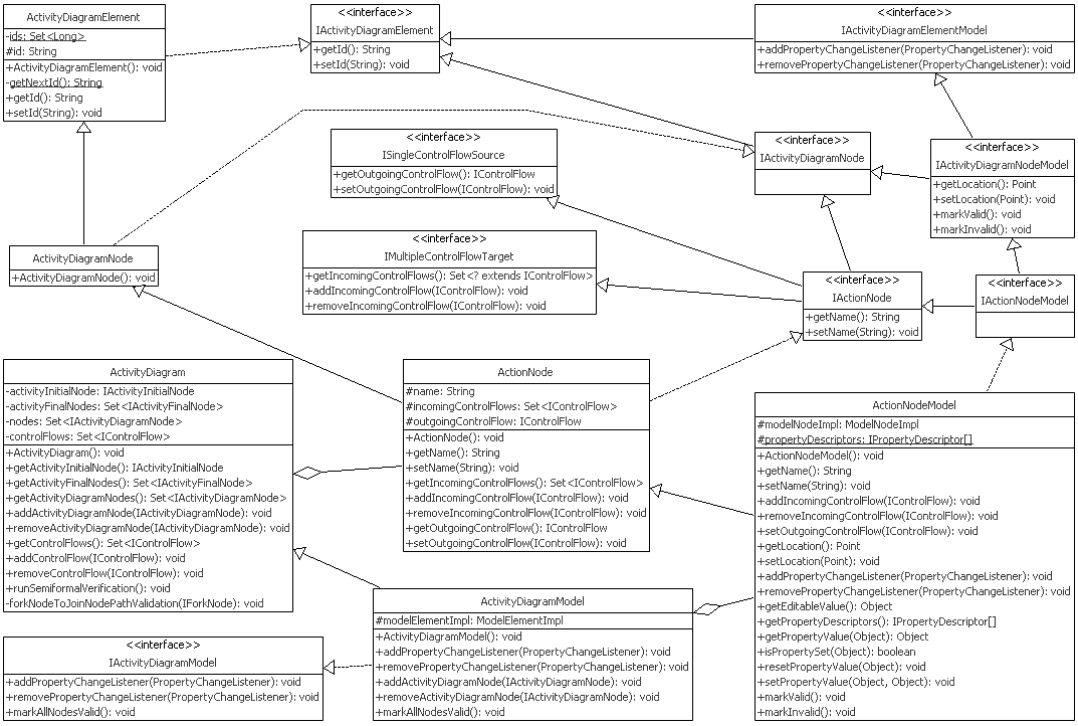
Příloha 3 – Class diagram: Petri net interfaces



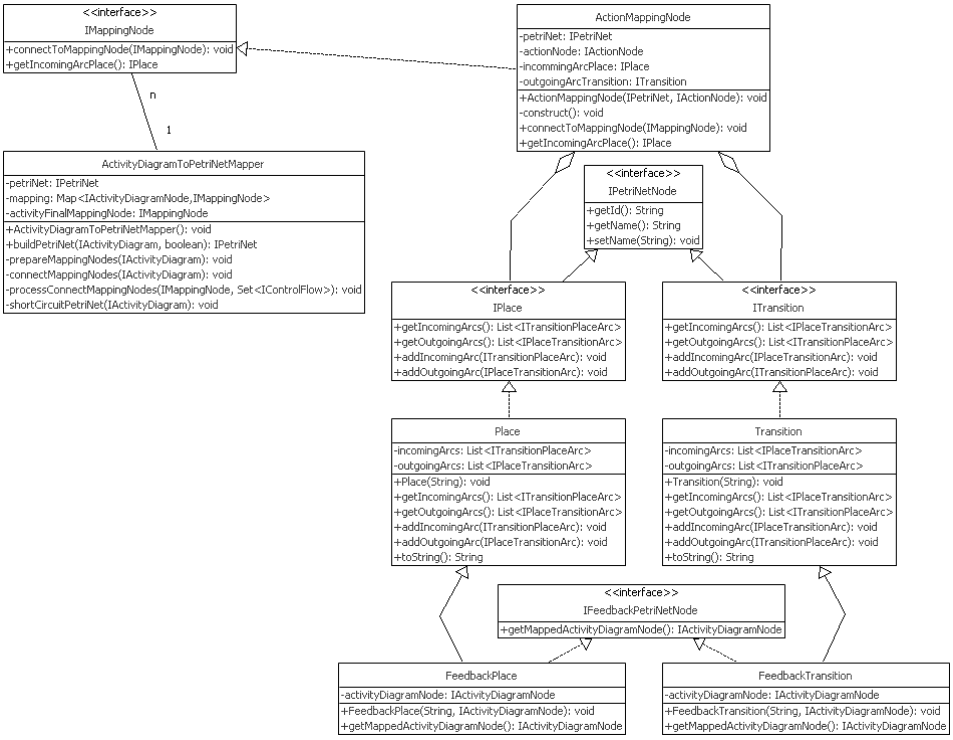
Příloha 4 – Class diagram: Petri net elements interfaces



Příloha 5 – Class diagram: UML 2.0 activity diagram mapping to Petri net element interfaces



Příloha 6 – Class diagram: action node model full hierarchy



Příloha 7 – Class diagram: action node model mapping hierarchy